

DTIC FILE COPY

AD-A230 498



DTIC
ELECTE
JAN 07 1991

D

D

Extraction and Measurement of Multi-Level
Parallelism in Productions Systems

THESIS

George Allen Sawyer
Captain, USAF

AFIT/GCE/ENG/90D-04

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

91 1 3 162

1

AFIT/GCE/ENG/90D-04

DTIC
ELECTE
JAN 07 1991
S D D

Extraction and Measurement of Multi-Level
Parallelism in Productions Systems

THESIS

George Allen Sawyer
Captain, USAF

AFIT/GCE/ENG/90D-04

Approved for public release; distribution unlimited

AFIT/GCE/ENG/90D-04

Extraction and Measurement of Multi-Level
Parallelism in Productions Systems

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

George Allen Sawyer, B.S.
Captain, USAF

14 December 1990

Approved for public release; distribution unlimited

Acknowledgments

This research was motivated by the significant problems encountered by past researchers in increasing the execution speed of expert systems using parallel computer architectures. By examining the expert systems to parallel computer architecture “transformation” from a theoretical basis, the salient factors affecting performance are revealed and characterized. The results indicate that the performance problem needs to be approached from both the application design and the parallel algorithm design aspects. Additionally, this research shows that the level of parallelism that can be extracted from an expert system may be multiplicatively increased through the use of multi-level parallelism (combining problem decomposition approaches). The use of multi-level parallelism offers the potential to increase the execution speed of expert systems applications one or more orders of magnitude in comparison with existing expert systems implementations on parallel computer architectures.

I would like to acknowledge members of the AFIT faculty, Dr. Frank M. Brown, Major William Hobart and most especially my thesis advisor, Dr. Gary Lamont for preparing me to undertake this project and for providing key guidance at many points during the research process. I am also indebted to many AFIT students, not only from my class, but other classes as well; especially, Captain R. Andrew Beard, Captain Edward Williams, Captain Jay-Evan Tevis, Major Bruce Conway, and Capt Jeffery Simmers for their assistance and support. Finally, I would like to acknowledge the invaluable infusion of advanced technical knowledge from Brian Milnes and Milind Tambe from Carnegie-Mellon University and Jay Labhart from Merit Technology.

I would like to dedicate this work to the people that have believed in me and made a difference: my parents, Carol J. Gage and Sheldon S. Sawyer Sr; my mentors, Major Bobby E. Roberts and Lt Col Rodney R. Rasmussen; and most importantly, my loving wife, Tina Maria Rice Sawyer.

George Allen Sawyer



A-1

Table of Contents

	Page
Acknowledgments	ii
Table of Contents	iii
List of Figures	x
Abstract	xiv
 I. Introduction	 1-1
1.1 The Expert Systems Solution/Problem	1-1
1.2 No Easy Solutions	1-2
1.3 Production Systems Versus Logic Programming	1-2
1.4 Review of Past and Current Research	1-5
1.5 Scope of The Research	1-8
1.6 Maximum Expected Gains	1-10
1.7 Overview of the Thesis	1-10
 II. Characteristic and Complexities of Production Systems	 2-1
2.1 Introduction	2-1
2.2 Production Systems: A General Definition	2-1
2.3 Production System Applications	2-3
2.3.1 A Generalized Search Example	2-4
2.3.2 Expert System Example	2-7
2.4 Production Systems as a Search Process	2-8
2.5 Time Complexity of Production Systems	2-11
2.5.1 Match Phase Time Complexity	2-13
2.5.2 Select Phase Time Complexity	2-14

	Page
2.5.3 Act Phase Time Complexity	2-15
2.5.4 Overall Time Complexity of Production Systems . .	2-16
2.6 Fast Matching Using Rete	2-18
2.6.1 A High Level Overview	2-18
2.6.2 Details of the Rete Algorithm	2-19
2.7 Why Rete is Better	2-24
2.8 Conclusion	2-25
III. Parallel Computing Concepts, Problems and Architectures	3-1
3.1 Introduction	3-1
3.2 Essential Concerns in Parallel Computing	3-1
3.2.1 Design and Development of Correct Parallel Programs	3-2
3.2.2 Parallel Program Performance	3-4
3.3 Decomposing a Problem/Composing a Design	3-7
3.3.1 Data Decomposition Techniques and Examples . . .	3-9
3.3.2 Control Decomposition and Generic Examples . . .	3-19
3.4 Parallel Computer Architectures	3-21
3.4.1 Choosing a Useful Subset of Parallel Architectures .	3-21
3.4.2 MIMD Architectures	3-24
3.4.3 SIMD Processor Array Architectures	3-29
3.5 Summary: Problems and Architectures, A Common Framework	3-31
IV. Quantifying Parallelism in Production Systems	4-1
4.1 Introduction	4-1
4.2 Sources of Parallelism	4-2
4.3 Application Parallelism	4-5
4.3.1 Measuring Application Parallelism	4-6
4.4 Agenda or Rule-Level Parallelism	4-6

	Page
4.4.1 Characteristics of Agenda Parallelism	4-8
4.4.2 Bounding Agenda Parallelism	4-10
4.5 Production Parallelism	4-11
4.5.1 Characteristics of Production Parallelism	4-13
4.5.2 Bounds on Production Parallelism	4-14
4.5.3 Deriving an Optimum Rule Partition	4-19
4.5.4 Measuring Production Parallelism	4-24
4.5.5 Affect of Agenda Parallelism on Production Parallelism	4-26
4.6 Node Parallelism	4-26
4.6.1 Approaches to Implementing Node Parallelism . . .	4-27
4.6.2 An Overall Model for Node Parallelism	4-31
4.6.3 Measuring Node Parallelism	4-34
4.6.4 Affect of Agenda Parallelism on Node Parallelism . .	4-45
4.7 Intra-Node Parallelism	4-46
4.7.1 Approaches to Intra-Node Parallelism	4-46
4.7.2 Bounding Intra-Node Parallelism	4-49
4.8 Summary	4-49
V. The STEPPS System Requirements and Design	5-1
5.1 Introduction	5-1
5.2 STEPPS Software Requirements Document	5-1
5.2.1 Requirements Introduction	5-2
5.2.2 The STEPPS System Model	5-3
5.2.3 STEPPS Functional Requirements	5-3
5.2.4 Software Development and Operating Environment .	5-7
5.2.5 Maintenance Information	5-7
5.3 The STEPPS System Design	5-8
5.3.1 The Application Parser Sub-system	5-8

	Page
5.3.2 Rule-Base Matrix Representation Sub-system	5-11
5.3.3 Parallelism Evaluation and Statistical Analysis Sub-system	5-12
5.3.4 Support Packages and Main Program	5-13
5.4 The STEPPS System Implementation	5-14
5.4.1 The Application Parser Sub-system	5-14
5.4.2 The Rule-Base Matrix Representation Sub-system .	5-19
5.4.3 Parallelism Evaluation and Statistical Analysis Sub-system	5-22
5.4.4 Support Packages	5-22
5.4.5 The STEPPS Main Program	5-23
5.5 Summary	5-23
VI. Investigating Parallelism Using STEPPS	6-1
6.1 Introduction	6-1
6.2 The Monkey and Bananas Problem	6-2
6.2.1 The OPS-5 MAB Implementation	6-4
6.2.2 The CLIPS MAB Implementation	6-4
6.3 The Digital Circuit Redesign System	6-5
6.4 The Bogus SDI Example Scenario	6-7
6.5 The Tourney Scheduling System	6-9
6.6 The Rubik's Cube Problem Solver	6-12
6.7 The Smoke1 Pollution Control Expert	6-14
6.8 The Robotic Air Vehicle System	6-16
6.9 Evaluation of STEPPS Performance	6-18
6.10 Synopsis of Investigation Results	6-19
6.11 Summary	6-21

	Page
VII. A Detailed Investigation of Agenda Parallelism	7-1
7.1 Introduction	7-1
7.2 Proving the Correctness of Agenda Parallelism	7-2
7.3 Implementation Issues in Agenda Parallelism	7-5
7.3.1 Modifying Conflict Resolution	7-6
7.3.2 Detecting Interference Between Rules	7-7
7.4 Choosing a Set of Rules to Execute	7-9
7.4.1 Maximizing the Number of Rules Executed	7-11
7.4.2 A Greedy Approach for Maximizing Rules Executed	7-17
7.4.3 Maximizing the Size of The Affect Set	7-19
7.5 Implementing Agenda Parallelism Using CLIPS	7-22
7.6 Parallel Rule Execution Case Studies	7-26
7.6.1 The Monkey and Bananas Problem	7-27
7.6.2 The SDI Scenario Production System	7-27
7.6.3 The Digital Circuit Redesign Expert System	7-28
7.7 Summary	7-30
VIII. Conclusions and Recommendations	8-1
8.1 Introduction	8-1
8.2 Review of Thesis	8-1
8.3 Research Results	8-3
8.3.1 Measuring Parallelism in Production Systems	8-3
8.4 Implementing Agenda Parallelism	8-5
8.5 Recommendations for Future Research	8-6
8.6 Summary	8-8

	Page
Appendix A. An Introduction to Logic Programming	A-1
A.1 Introduction	A-1
A.2 An Example Problem in Prolog	A-1
A.3 Resolution and Refutation in Prolog	A-7
A.4 Parallelism in Prolog Problem Solving	A-9
Appendix B. A Discussion of the UNITY Approach	B-1
B.1 Introduction	B-1
B.2 UNITY Concepts and Notation	B-1
B.3 A UNITY Program Example	B-4
B.4 Proving UNITY Program	B-5
B.5 Mapping to A Parallel Architecture	B-8
B.6 Implementing Odd-Even.c on a Parallel Computer	B-9
Appendix C. Detailed Discussion of Search Algorithms	C-1
C.1 Introduction	C-1
C.2 A* Search Analysis and Implementation	C-1
C.2.1 A* Search Background	C-1
C.2.2 The Optimal Partitioning Algorithm Implementation	C-2
C.3 Genetic Algorithm Based Search	C-5
C.3.1 Genetic Algorithms Background	C-5
C.3.2 The Genetic Partitioning Algorithm Implementation	C-12
Appendix D. STEPPS Users Guide	D-1
D.1 Introduction	D-1
D.2 Using STEPPS	D-2
D.2.1 STEPPS Inputs	D-2
D.2.2 STEPPS Program Outputs	D-7
D.3 STEPPS Cautions and Limitations	D-8

	Page
D.3.1 Static Data Structures Limitations	D-8
D.3.2 Rule Complexity Limitations	D-8
D.3.3 Language Compatibility Cautions	D-9
D.4 Obtaining and Building STEPPS	D-10
Appendix E. STEPPS System Source Code	E-1
E.1 Introduction	E-1
E.2 File Names and Contents	E-1
Appendix F. Multi-CLIPS System Source Code	F-1
F.1 Introduction	F-1
Vita	VITA-1
Bibliography	BIB-1

List of Figures

Figure	Page
1.1. Different Approaches to Expert Systems	1-3
1.2. Some Important Event in the History of Parallel Production Systems .	1-6
2.1. The Production Cycle	2-3
2.2. Graphical Representation of the Problem	2-5
2.3. The Dimes.clp Program	2-6
2.4. Hierarchy Chart for Concrete Master	2-8
2.5. The "Concrete Master" Expert System Example	2-9
2.6. Production System Search Space	2-12
2.7. A Rule to WM Element Match	2-13
2.8. Levels of Reasoning for a Simple Expert System	2-17
2.9. An Example Two Rule Rete Network	2-21
2.10. Example of an <i>and</i> Node Activation	2-23
3.1. Overview of the UNITY Approach	3-3
3.2. Matrix Multiplication Example	3-10
3.3. Matrix Multiplication UNITY Program	3-11
3.4. An Example Matrix Multiplication Time/Space Lattice	3-12
3.5. Example solution for the 5-queens Problem	3-14
3.6. Search Tree for the 4-Queens Problem	3-15
3.7. N-Queens BFS UNITY Program	3-16
3.8. A Generic Static Control Decomposition Example	3-19
3.9. A Generic Dynamic Control Decomposition Example	3-20
3.10. High-Level Taxonomy of Parallel Architectures (24:6)	3-22
3.11. A Useful Subset of Parallel Architectures	3-23

Figure	Page
3.12. Architecture of a Generic Multi-computer (73:280)	3-24
3.13. Distributed Memory MIMD Topologies (30:23) (24:10)	3-25
3.14. Architecture of a Generic Multi-Processor (73:280)	3-26
3.15. Architecture of a Generic Multi-Processor (73:158)	3-28
3.16. General Structure of a SIMD Array Architecture	3-30
3.17. A General Common Framework for Problems and Architectures (31:953)	3-32
4.1. A Hierarchy of Problem Decompositions	4-2
4.2. UNITY Program for a Simple Production System Shell	4-4
4.3. Pilots Associate, Top Level Architecture (78:4)	4-5
4.4. A Simple Comparison of Sequential and Parallel Rule Firing	4-7
4.5. The Soar Formalism Interpreter Cycle (36:44)	4-9
4.6. UNITY Program for a Simple Production System Shell	4-10
4.7. Production Parallelism	4-12
4.8. UNITY Program for a Simple Production System Shell	4-14
4.9. Example Ad-hoc Allocation of Rules to Processors	4-16
4.10. Selected Rule Partitioning Results From Ofizer's Research (59:98) . .	4-20
4.11. Rule-Processor Allocation Table and Rule-Antecedent Cross Reference Table	4-23
4.12. Optimum Partitioning of the Rule Base in Figure 4.9 for 3 Processors .	4-23
4.13. Optimum Partitioning of the Rule Base in Figure 4.9 for 2 Processors .	4-24
4.14. The CMU Production System Machine Implementation (40:106)	4-28
4.15. The METE Algorithm Parallel Inference Architecture (51:103)	4-29
4.16. Organization of the NON-VON Machine (44:242)	4-30
4.17. High-Level View of Proposed Multi-Computer Mapping (75:7)	4-31
4.18. Scheduling Constant Test Nodes (36:76)	4-32
4.19. A Parallel Two-Input Node Activation (36:49)	4-33
4.20. An Example for Measuring Node Parallelism	4-35

Figure	Page
4.21. An Arbitrary Rete Network Fragment	4-42
4.22. Proposed Intra-Node Parallelism Implementation Using Attached Processes	4-47
4.23. Proposed Intra-Node Parallelism Implementation Using a SIMD Architecture	4-48
5.1. The STEPPS System Model	5-2
5.2. STEPPS High-Level Data Flow Diagram	5-9
5.3. STEPPS Complete Object-Oriented Design	5-10
5.4. An Abstract View of the List Data Type	5-14
5.5. Low-Level Data Flow Diagram for the Parser Sub-System	5-17
5.6. Application For Illustrating \mathcal{B} Matrix Reduction	5-20
5.7. Original (left) and Reduced (right) \mathcal{B} Matrices	5-21
6.1. A Visual Representation of the MAB Problem	6-3
6.2. Example Digital Circuit Redesign Transformation Rule	6-5
6.3. Example Digital Circuit Redesign Simplification Rule	6-5
6.4. Production Parallelism Estimates for the DCR System	6-7
6.5. A Example Schedule Produced by Tourney	6-10
6.6. Production Parallelism Estimates for the Tourney System	6-12
6.7. An Example of Splitting Two Input Rete Nodes (75:16)	6-13
6.8. The Robotic Air Vehicle (RAV) System Architecture (57:1327)	6-16
6.9. Production Parallelism Estimates for the RAV System	6-18
7.1. An Example of Interfering Rules in a Production System	7-2
7.2. Visual Representation of the Example Problem	7-14
7.3. Adjacency Matrix for the Example Problem	7-14
7.4. The "T" Matrix for the Example Problem	7-15
7.5. SCP Tableau for the Example Problem Using Unity Costs	7-16

Figure	Page
7.6. SCP Search Tree Generated for the Example Problem	7-18
7.7. The CLIPS High-Level System Architecture	7-23
7.8. Agenda Parallelism Execution Traces for Bogus.kb	7-29
7.9. Agenda Parallelism Execution Traces for the DCR System	7-31
8.1. A Visual Overview of the Thesis Document	8-2
A.1. A Comparison of Logic Syntax	A-1
A.2. Family Tree for the Example Prolog Problem	A-3
A.3. A Short Session With A Prolog Interpreter	A-6
A.4. Using Resolution To Solve the Aunt Problem	A-8
A.5. A Representation For Part of The "Aunt" Problem	A-10
B.1. UNITY Example Program 1 For Odd-Even Transposition Sort (15:438)	B-4
B.2. UNITY Example Program 2 For Odd-Even Transposition Sort (15:439)	B-4
B.3. Two Example Parallel Odd-Even Transposition Sorts	B-6
B.4. Distributed-Memory UNITY Program For Odd-Even Transposition Sort	B-10
C.1. The A* Search Rule-to-Processor Partitioning Algorithm	C-5
C.2. Genetic Algorithm Example (33:16)	C-8
C.3. Selection Roulette Wheel (33:11)	C-9
C.4. Genetic Algorithm Example (cont) (33:17)	C-10
C.5. Example Crossover Operation	C-10
D.1. The "awkclips" Rule Weight Processing Program	D-6
D.2. The "awkops5" Rule Weight Processing Program	D-6

Abstract

Productions systems provide a flexible and powerful means for expressing and solving problems using high-level reasoning approaches. Unfortunately, applications based on the production system paradigm are very compute intensive and therefore execute very slowly on conventional computer architectures. Lack of execution speed therefore limits the use of production systems in many time sensitive and complex tasks. Researchers have demonstrated that increasing the execution speed of production systems through the use of small scale parallel computer architectures is possible, however demonstrated execution speed is still several orders of magnitude below the desired level of performance. Past research indicates that there exists an irrevocable tie between the specific production system application and the level of parallelism inherent in that application. To date, this parallelism has been quantified only in very loose terms based on specific problem decomposition and implementation on a specific parallel architecture.

This research examines many of the possible approaches to extracting parallelism from productions systems and quantifying the level of parallelism in specific applications. This research also investigates previously unexplored aspects of several important decomposition approaches. The effect that combining different decomposition approaches (applying multi-level parallelism) has on the level of available parallelism is also considered. Knowledge gained from an analysis of parallelism in different production systems in turn provides a better understanding of how parallel computer architectures can be used to increase the execution speed of production systems. The software products developed to support this research are Software Tool for Evaluating Parallelism in Production Systems (STEPPS) and a modified version of the C-Language Integrated Production System (CLIPS) called Multi-CLIPS.

STEPPS is the primary software product and is used to actually measure the level of inherent parallelism in a number of applications. The purpose behind using STEPPS to measure the level of parallelism in these applications is two-fold. First, the measurements are able to determine which problem decomposition approach (or combination of

decomposition approaches) is most likely to provide the greatest possible speed-up for a given application. Combining the results of a number of applications reveals which problem decomposition approach provides the best possible speed-ups over a large number of range of different applications. Second, the measurements provide a basis for determining which applications are amenable to speedup. The characteristics and overall design of these systems can then be used as guide to aid developers in designing and implementing production systems capable of achieving greater performance on parallel architectures.

The Multi-CLIPS software product is a modified production system shell supporting the limited investigation of parallel decomposition techniques that can only be evaluated using dynamic analysis techniques. Theoretical analysis of previously unexplored aspects of these parallel decomposition techniques supports the modification of the original CLIPS shell. Multi-CLIPS proves useful in not just examining the level of parallelism in different production systems applications. The Multi-CLIPS system is also an important source of information on unforeseen problems relating to the implementation of some previously unexplored, but promising, parallel decomposition techniques.

Overall, the measurement of parallelism (both static and dynamic) in different applications provides a better understanding of the general problems associated with applying larger scale parallelism to production systems. Finally, the insight gained from this research serves to guide the future development of efficient parallel production system shells and applications capable of maximum use of these shells.

Extraction and Measurement of Multi-Level Parallelism in Productions Systems

I. Introduction

1.1 The Expert Systems Solution/Problem

Despite the computational power of modern computers, there are still a significant number of tasks, such as piloting an aircraft, that trained humans do better than computers. Attempts to apply traditional computer programming approaches to these types of problems typically fail due to the complexity and unmanageability of the computer code (program) (41:1). Expert systems are computer programs which emulate the decision making capabilities of a human expert in a specific field of knowledge. They are successfully addressing many of the problem areas by applying unique high level reasoning approaches derived from the computer science field of Artificial Intelligence. Although expert systems are capable of solving many of the problems that defy traditional programming approaches, they do have problems. One the most significant problems is speed of execution (52:27).

Douglass observes that typical expert systems in commercial use today are relatively small, but they execute very slowly even when using powerful computers (23:70). The viability of many projected expert systems applications rely on increasing their relative execution speeds significantly. Laffey and his co-authors cite numerous examples of expert system designs under consideration for monitoring and control tasks traditionally requiring trained human operators. Although these applications are theoretically possible, current single processor computer systems are not fast enough to support them (52:29-35). The U.S. Strategic Computing Program targets execution speeds of 12,000 rules per second for expert systems applications containing 30,000 rules. In contrast, today's typical expert systems performance is only 20 rules per second on relatively small applications of 1000 to 2000 rules. The ratio between currently available execution speeds and the target execution rates represents a speed-up of 600 times. Obtaining the required increases in execution

speed, if it is possible, will likely require implementing expert systems on large scale parallel computers (23:70).

1.2 No Easy Solutions

The greatest obstacle in increasing the execution speed of expert systems applications using parallel computers lies within the applications themselves. Several researchers cite the irrevocable ties between the level of parallelism inherent in a specific application and the manner in which that specific application is constructed (36:5) (7:1) (22:20). Clearly, increasing the parallelism in a specific application depends on the application's designer and his/her skill in extracting parallelism from the original problem. In order to maximize opportunities for parallelism, the designer needs to know: What level of parallelism currently exists in an application under development? This in turn raises another important question: Is it possible to establish, with any certainty, the inherent parallelism in a given expert systems application? My research intends to answer this second question by deriving a flexible set of metrics¹ for quantifying the level of parallelism that can be extracted from a given application. If successful, the performance metrics will provide the developer with a priori knowledge on how to best apply the existing level of parallelism inherent in an expert systems application. The developer will also be aware of whether an application has the potential to meet its processing speed requirements through the use of parallel processing.

1.3 Production Systems Versus Logic Programming

Most expert systems can be categorized as either forward chaining or backward chaining depending on the way in which they carry out the search for a problem solution. The forward chaining (or data-driven) method begins with a set of assertions about a problem and then applies logical rules in the attempt to find a solution. One example of forward chaining might involve choosing the next move in a game of chess. Production systems such as Official Production System Language, version 5 (OPS-5) provide a means for directly

¹In this context, metrics is intended to convey some means of accurate measurement.

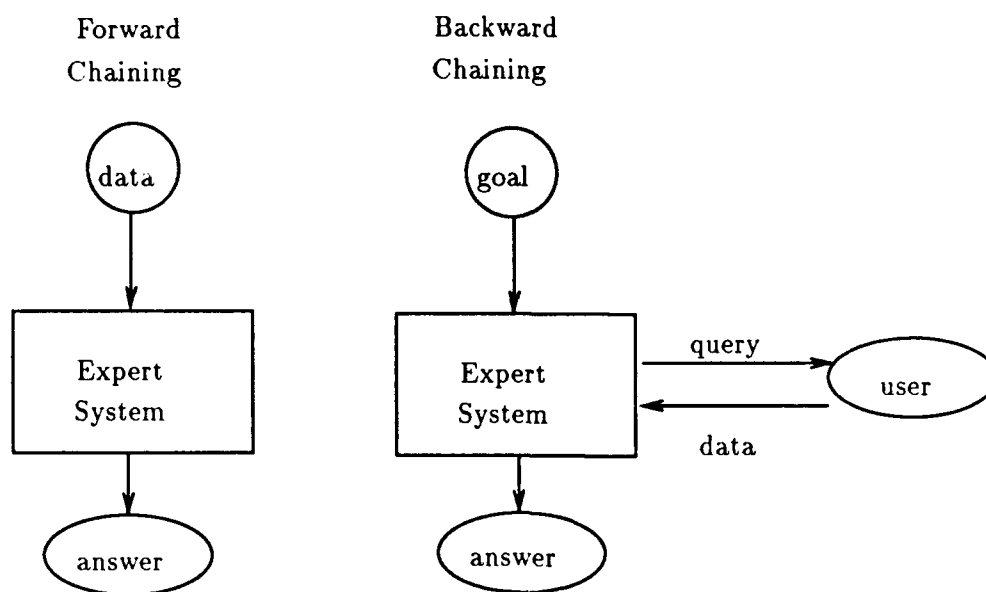


Figure 1.1. Different Approaches to Expert Systems

implementing forward chaining expert systems. On the other hand, backward chaining (or goal driven) expert systems begin with a statement of the solution and determine if this solution is attainable from the available facts. Luger and Stubblefield present backward chaining example of solving a maze problem by starting at the finish and tracing backward to the start (54:88). Prolog² based systems such as Back-Chainer, version 3 (BC-3) provide a straightforward means for implementing backward chaining expert systems. The research project chooses to concentrate primarily on the production system approach to expert systems because of their flexibility and (relative) time efficiency.

Production systems possess the potential for both the speed and the flexibility required for the successful implementation of expert systems for many complex and time-critical, military and civilian applications. Douglass notes that given an appropriate set of rules, production systems are capable of producing backward chaining as well as forward chaining expert systems (23:73). Luger and Stubblefield cite the ability of production systems to model many aspects of the human cognitive process (54:114). The "bi-directional"

²Prolog stands for Programme Logique, a logical programming method based on the resolution method defined by Robinson in 1965 and first implemented by Colmerauer in 1972

and knowledge representation "power" of production systems combine to make production systems a flexible tool for creating expert systems. Production systems are also useful in developing high performance expert systems because of their particular characteristics: Jackson notes that most production systems use control methods that eliminate the time intensive backtracking (looking for alternate solutions) found in most logic programming systems (49:87). Douglass observes that the control process of production systems are much simpler than for logic programming (23:76) which generally results in simpler and thus faster implementations. Jackson also observes that efficient algorithms for implementing production systems exist that have no logical equivalent in Prolog based systems (49:88). Chapter 2 presents significantly more information on the characteristics and complexities of production systems. For the sake of completeness and to answer possible questions related to logic programming, Appendix A provides a brief discussion on the Prolog approach to expert systems.

Although Prolog based expert systems have proven tremendously useful in many application areas, they do not appear to possess any attributes that make them better than production system based expert systems. The one notable exception to this assertion is in the design of simple advisor type expert systems. As demonstrated in Appendix A, Prolog based systems have a built-in query handling process that aids substantially in the construction of advisor type systems. Evidence suggests that parallelism in Prolog based expert systems is no greater than (and may be less than) parallelism in production systems (23:80-81). Additionally, Prolog based expert systems are often used in advisor type roles that require significant interaction with a user (23:79). In contrast, as Figure 1.1 shows, forward chaining systems typically require very little user interaction except when specifying the initial data. Each time the expert systems requires communication with the user, processing effectively halts until the user enters the required information. Unless a significant amount of processing is accomplished between queries for user information, using parallelism to increase execution speed will mean very little. Douglass also observes that required user-system interaction imposes some severe constraints on how parallelism can be exploited by Prolog based systems (23:79). Although there are possibilities for exploiting parallelism in Prolog based expert systems, most of these possibilities remain

relatively unexplored.

1.4 Review of Past and Current Research

Expert systems based on the production systems paradigm draw from a wide background of research that begins as early as 1943 and extends to today. Figure 1.2 presents some of the more important points in past research with direct applicability to the research effort defined by this document. The remainder of this section is devoted to a brief discussion of the research behind the important events in this figure.

When Post originated production systems in 1943, the concept of artificial intelligence (AI) did not exist, let alone the realization of AI concepts such as expert systems. The theoretical aspects of production systems as defined by Post are the basis for the work of Markov who provided a more useful framework for the use of production systems model of computation (32:16). The use of production systems in the field of AI was popularized by Newell and Simon in the late 1950s and early 1960s, but it was not until 1965 that work began on the first acknowledged expert system. The first expert system, DENDRAL, was used to aid in determining the composition of particular organic molecules using Mass Spectrometer Data³. The success of DENDRAL and other expert system experiments spurred the development of larger and more complex expert systems for commercial use (32:15). However, as discussed earlier, this advancement has cost dearly in terms of the performance of these systems (52:27). Since the late 1970s, researchers have looked toward algorithm improvement and the use of parallel computer to the performance of expert systems. The Rete algorithm by Forgy represents a significant increase in expert system processing efficiency, but improvements since that time have not been nearly as significant. Although the continued refinement of algorithms is important, the emphasis for increasing execution speed now appears to rest primarily on the use of parallel expert systems.

A parallel expert system represents the implementation of a specialized expert system control structure on a parallel computer. Most expert systems consist of a database of knowledge about a given problem domain and a control structure, also known as an

³See (49) for an excellent high-level description of the DENDRAL program and how it works

- 1943 - Post formally defines the production system computation model (32:12).
- 1957 - Production systems used by Newell and Simon to emulate human problem solving behavior (32:12).
- 1965 - Work begins on the first expert system at Stanford (32:12).
- 1974 - Researchers at Stanford University develop EMYCIN, the first expert system shell (32:12).
- 1977 - Forgy implements an early OPS shell on the Illiac IV parallel computer (36:4).
- 1980 - Forgy's Rete fast match algorithm is incorporated into the OPS series of shells (29:17).
- 1983 - Forgy and Gupta perform baseline measurements on a number of production systems (37)
- 1984 - Stolfo and Ishida first propose the concept and framework for production parallelism with parallel rule firing (48).
- 1985 - Work at Columbia University begins on DADO and Non-Von, two parallel computers designed specifically for executing production systems applications (44) (72).
- 1985 - Work completed on the first version of OPS-83, a shell that runs in compiled mode instead of interpreter (28).
- 1986 - Simulations by Gupta indicate that speed-ups of up to 20 times can be achieved using parallel computers (36).
- 1986 - Ofrazier completes a study that indicates that only limited parallelism (5 to 6 times) is available using production parallelism (59).
- 1987 - The para-OPS-5 shell is implemented on an Encore Multimax parallel computer and achieves a speed-up of 12.4 fold (40).
- 1989 - New model for parallel rule execution to increase processing speed on parallel computers is proposed by Dixit and Moldovan (22).
- 1989 - Merit Technologies achieves a speed-up of 29 times using a special version of the Rete algorithm on a BBN Butterfly parallel computer (51).
- 1989 - HyperCLIPS expert system shell for the Intel Hypercube parallel computer reveals more difficulties in implementing parallel production systems (41).
- 1990 - Merit Technologies continuing development of an expert system shell for the Connection Machine parallel computer (51).

Figure 1.2. Some Important Event in the History of Parallel Production Systems

inference engine or shell, for applying the knowledge to solve a specific problem. Expert system knowledge bases are mostly application-specific, but inference engines are often totally independent of these knowledge bases and can be used for numerous expert system applications (23:71-73). A parallel computer is a computer that contains multiple processing elements, each of which is capable of independently manipulating separate data streams at the same time (in parallel). The parallel expert systems problem involves effectively embedding the inference engine control structure on the separate processing elements of a parallel computer. Done effectively, this embedding process should yield a parallel computer capable of executing nearly any expert system application significantly faster than a single processor computer (41:10).

An intelligent approach to a difficult problem like parallel expert systems requires an in-depth understanding of the problem. Although expert systems control structures are very similar, knowledge bases vary significantly depending on the specific application of the program. In 1983, Gupta and Forgy studied six expert systems in order to understand their behavior and characteristics. The six expert systems, developed by Carnegie-Mellon University (CMU) for various customers and research projects, represent a good cross section of available expert systems. From measurements on these expert systems, Gupta and Forgy conclude that parallelism in expert systems is limited to approximately 35 fold. They also determined that this level of parallelism does not increase as the overall size of a given expert system increases (29:29). Gupta's analysis of six additional expert systems confirms these findings. In fact, Gupta suggests that parallelism may be limited to an average of 26 fold by inconsistencies in average knowledge bases (36:162).

Since the first parallel production systems experiments by Forgy in the mid-1970s, many researchers have focused on implementing the parallelism known to exist in production systems. Due to the limitations of early parallel computers, Forgy notes significant difficulties in achieving performance greater than that of sequential computers (36:4). As the "state-of-the-art" in parallel computing advanced significantly in the 1980s, numerous attempts were made to increase execution speed using many different types of parallel computers. In the mid-1980s, Columbia University built several parallel computers designed specifically for executing production systems, however the most successful

experiments thus far have used commercially available parallel architectures. Researchers at Carnegie-Mellon University have achieved speed-ups of more than 12 times on standard expert system applications (40:95). Merit Technologies have achieved speed-ups of almost 30 times on an expert system that was specially designed for parallel execution (51:1).

Past and current research reveals two important aspects of parallelizing expert systems application: First, it shows that parallel processing can be used to increase the execution speed of almost any expert system application. Second, it shows the feasibility of developing a given expert system application with greater than average parallelism. The presence of both of these factors give credence to the possibility of achieving at least some of the speed-up targeted by the US Strategic Computing Program.

1.5 Scope of The Research

This study concerns only an assessment of parallelism in production systems with the capability to apply similar techniques to other expert system applications. The thrust of the research is to investigate accurate prediction of specific expert systems applications performance using parallel processing. The research is primarily algorithm oriented and does not specifically investigate hardware issues beyond the affects of different parallel computer architectures on achievable performance. Additionally, this research does not specifically attempt to produce any highly parallel expert systems applications or commercial quality software products. Specifically, the scope of this research encompasses the following:

1. Investigate the sources of parallelism in expert systems and the constraints placed on this parallelism by specific applications.
2. Construct a rudimentary software tool for assessing the level of parallelism in a given expert systems application based on the metrics developed under part 1.
3. Evaluate the accuracy and shortcomings of the parallelism measurement metrics and make refinements to the metrics if possible.

The first phase of this research involves developing a sufficiently accurate but flexible model of general expert systems behavior and how the level of parallelism relates to appli-

cation specific characteristics. The study resulted in a generalized procedure for assessing inherent parallelism in a specific expert system application. The second research phase develops a software tool that automates the generalized procedure developed during the first phase. Given the relative size of even simple expert system applications, performing any useful analysis by hand is not practical, hence the development of the software tool. The final research phase compares the performance predictions of the software tool with the actual performance of different expert system applications hosted on specific parallel computers. Based on a comparison of predictions with actual results plus other factors, the research also examines the potential for accurately evaluating the level of parallelism in different production systems applications. The following paragraph explains, in more detail, the approach used in developing the expert system model.

Developing a model that is both flexible and accurate requires a modular type design that allows a number of alternate approaches corresponding to specific implementation decisions. The basis of the model will rely on an analysis based on the *surface characteristics* of a specific expert systems application possibly augmented with very limited *dynamic characteristics* analysis. Surface characteristics refer to the textual features of a specific expert systems application; they can be determined simply by analyzing the application computer code. Dynamic characteristics on the other hand are attributes that can be determined only by actually executing a specific expert system application. Surface characteristics are not influenced by any subsequent implementation details, therefore they provide the most generally applicable information about a given application. Dynamic characteristics are capable of providing significantly more information about how a certain sets of problem data are handled by the system, but may not reflect the overall characteristics of the given application unless the expert system is used to solve a number of different problems. The model for assessing parallelism specifically avoids executing the actual program, but will take advantage of any implementation specific knowledge that is available prior to execution.

1.6 *Maximum Expected Gains*

Gupta and Forgy's research indicates that significant amounts of parallelism exist in most expert systems; they state that this parallelism translates to a potential for 26 to 35 fold increases in execution speed when executing production systems on parallel computer architectures as opposed to sequential ones. However, 26 to 35 fold increases in execution speed hardly constitutes large scale parallelism. The alternatives are to accept this level of parallelism and attempt to realize as much of it as possible or attempt to create expert systems with greater degrees of inherent parallelism. Current parallel expert system implementations are close to achieving the 26 to 35 fold speed-up already. It is probably unrealistic to assume that advances in hardware technology will yield the additional processing power required to make a number of the proposed time-critical expert systems described by Laffey (52:71-73) a reality any time in the near future. On the other hand, *creating applications with larger amounts of parallelism is not likely to be an easy task for the application developer*. In his highly regarded article *No Silver Bullet*, Brooks stresses the importance of providing automated tools to aid the designer in managing complex design tasks (12:16). A tool to assess the the level of parallelism in a given expert system application can be considered as a first step in providing the developer with necessary tools. In the near term, this tool can be used to determine if a given expert system application is worth parallelizing and if it is, what the most promising methods are. In the long term, it will also provide a basis for a better understanding of the problems associated with parallelizing expert systems. Labhart projects that, in the long term, this understanding may provide the basis for tools capable of increasing the parallelism in a given application without affecting the functionality of that application (7:5). Applying long term gains in understanding of the problem probably holds the greatest promise for achieving the desired target performance values and making more complex expert systems applications a reality.

1.7 *Overview of the Thesis*

This research attempts to tie together the concepts of production systems with parallel processing to guide the development of expert systems applications capable of producing

efficient results when implemented on specific parallel computer architectures. The thesis attacks two main threads of the problem separately and then brings them together as a single coherent concept. Chapter 2 examines the characteristics and complexities of production systems in detail, including how they are used to solve problems. The concepts and problems associated with parallel computing are covered in Chapter 3; this chapter also covers the characteristics of different classes of parallel computers. Chapter 4 brings the concepts of production systems and parallel computing together to examine how parallel computing can be used to speed-up the execution of production systems. Chapter 5 describes the design and implementation of the software tool for evaluating parallelism in production systems applications (STEPPS) based on the concepts covered in Chapter 4. Chapter 6 presents the results of using the STEPPS system to analyze a number of different production system applications. Chapter 7 presents a detailed investigation of parallel rule execution in production systems, a topic that has not previously been considered in significant detail; this investigation includes an actual implementation of this type of parallelism in the form of the Multi-CLIPS system. Finally, Chapter 8 reviews the important points of the thesis, offers some important observations about the usefulness and accuracy of the STEPPS system and suggests directions for future research. Appendices on the logic programming approach to expert systems (Appendix A), the UNITY approach to parallel program design (Appendix B) and detailed discussion of the search and optimization algorithms used (Appendix C) present supplementary information on important concepts covered in the thesis. Additional information on the STEPPS system is contained in the STEPPS Users Guide (Appendix D) and the STEPPS program source code (Appendix E). The source code for the Multi-CLIPS program is contained in Appendix F. Due to the size of source code appendices (E and F), they are actually contained in a separate document.

II. Characteristic and Complexities of Production Systems

2.1 Introduction

The production system is a model of computation that has proven very useful for modeling the human cognitive process and for implementing generalized search processes. Production systems provide a natural and flexible means for representing and solving problems, especially problems that use "rule of thumb" type approaches. However, applications based on production systems run very slowly relative to applications implemented in more traditional (imperative) programming languages. This chapter presents a description of production systems and gives some simple examples of how they are used to solve problems. After describing production systems as a search process, this chapter provides information, along with examples, on why production systems are so inefficient. Finally, this chapter presents a means for significantly increasing the efficiency of production systems, the Rete match algorithm.

2.2 Production Systems: A General Definition

The productive system paradigm was originally proposed by Post in 1943 as a formal computational model, but it has found more important roots in the field of Artificial Intelligence (AI). Newell and Simon (Carnegie-Mellon University) are primarily responsible for bringing production systems into the realm of AI. In their book, "Human Problems Solving", Newell and Simon showed how production systems could be used to encode the problem solving capability of human experts (54:132). These systems emulating the problems solving capabilities of human experts are now known as *rule-based expert systems*. Production systems do not completely model the cognitive behavior of humans, but they have a number of features that make them attractive tools for building expert systems. Production systems provide a generalized search framework that can be used for tasks other than modeling human cognitive behavior. Generalized state-space search using different algorithms is another typical application of the production system paradigm. Solving the classical 8-puzzle or, more recently, the Rubik's cube problem are examples of state-space search problems.

In their most generalized form, production systems carry out pattern-directed search and replacement using rules that are based on a subset of first-order predicate logic. The execution of various rules modifies a set of facts which describes the current state of a specific problem (or search) (42:2). Luger and Stubblefield provide an expanded definition of a production system by describing their basic components (54:131)

- A set of production rules collectively known as the rule base. Each production rule is in the form of a condition-action pair and represents a single “chunk” of problem solving knowledge for a specific domain. The condition part of the rule (also known as the left-hand-side (LHS)) contains a pattern of one or more condition elements (CEs) (i.e. logical predicates) that must be satisfied using the contents of working memory (WM) in order to apply the rule. The action part of the rule (also known as the right-hand-side (RHS)) specifies an action to be carried out if the LHS of the rule is satisfied. An example rule is :

```
(if
    (aircraft assigned-to ?runway-num) and
    (?runway-num status clear)
then
    (add (aircraft cleared-to land)))
```

- A set of facts, collectively known as working memory (WM) that describes the current “state of the world”. Each fact is a pattern that can be matched against the CE in the LHS of different production rules. The contents of working memory are typically manipulated through the action specified by the RHS of matched production rules. One of the most common ways of representing facts is as an object with an attached list of attribute-value pairs. An example of a working memory (WM) fact is:

```
(aircraft call-sign United-223)
```

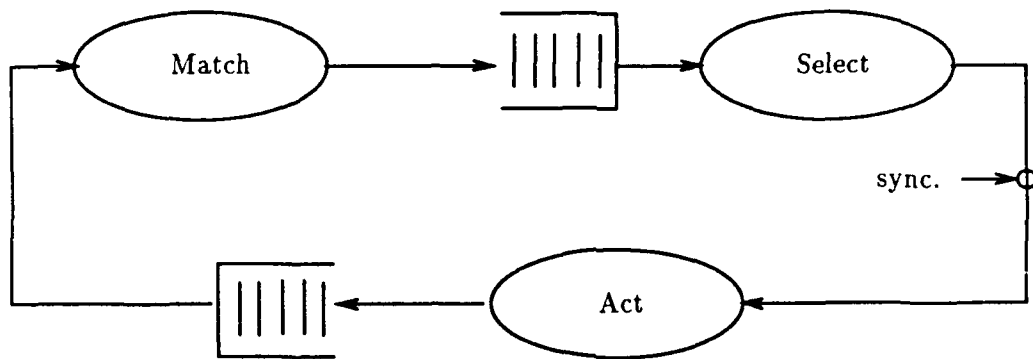


Figure 2.1. The Production Cycle

- A recognize-act cycle (also known as the match-select-act cycle) for controlling the state-space search as illustrated by Figure 2.1. At initialization, the working memory (WM) is loaded with a set of facts describing the start state. The recognize-act cycle then attempts to satisfy each of the rules in the rule base by matching the CEs in each rule's LHS with the facts in working memory (WM). All rules satisfied during this "match" phase are eligible for execution and are placed in a conflict set. Following completion of the match phase, the interpreter chooses one of the rules in the conflict set based on one of a number of conflict resolution techniques. The interpreter executes the actions specified by the RHS of the chosen rule. Unless an explicit stop condition is encountered, the interpreter returns to the "match" phase and the cycle continues.

2.3 Production System Applications

So far this chapter has presented a high level definition of production systems and describes very basically, how they work. This definition is useful information, but it really doesn't show how problems are actually solved using production systems. This section presents two simple production system applications to "close the gap" between theory and actual implementation. The first application solves a simple problem using a generalized search process. The pedagogical problem addressed in the search problem involves re-ordering a linear arrangement of dimes and nickels according to a set of specific rules. The second application involves applying simple human problem solving behavior to make an

optimizing decision. More specifically, the second example is a rule based expert system that helps a construction manager to create a proper concrete mix. Together, these two examples provide some indication of how production systems are used to solve significantly more complex problems.

2.3.1 A Generalized Search Example The first problem is a state-space-search based on the following problem description by Brown (13:1)

"A linear row is divided into $2k + 1$ squares containing k nickels and k dimes all in separate squares. In the initial configuration, all of the nickels are at the left and all of the dimes are at the right. We represent this for the case $k = 2$ by the list (nickel, nickel, space, dime, dime). The problem is to find a sequence of allowed moves which, when completed, will put the dimes at the left and the nickels at the right; thus the goal state is represented by the list (dime, dime, space, nickel, nickel). The allowed moves are:

1. slide a nickel to the right into an empty space
2. slide a dime to the left into an empty space
3. jump a nickel to the right over a dime and into an empty space
4. jump a dime left over a nickel and into an empty space".

Figure 2.2 presents a graphical representation of the problem to be solved including the initial state, the goal state and all of the legal moves as stated in the problem description. This is a relatively easy problem to solve using modern production systems languages such as the C-language Integrated Production System (CLIPS) written by Johnson Space Center. Figure 2.3 contains the CLIPS source code for solving this problem; readers should understand that this example is only one of many possible ways for solving this problem. The initial problem state as well as the goal state are asserted as initial facts. Most of the remaining rules are applied to the contents of Working Memory (WM) in a non-deterministic order to create new states. In these facts, the variables (x,y) preceded by "\$" indicate multiple field variables that contain 0 or more single field variables. The fact labeled "found-a-solution" is fired only when one of the states in Working Memory matches the specified goal state. With a little more elaboration, it would be possible for the program to print-out the actual steps taken to get from the initial to the goal state.

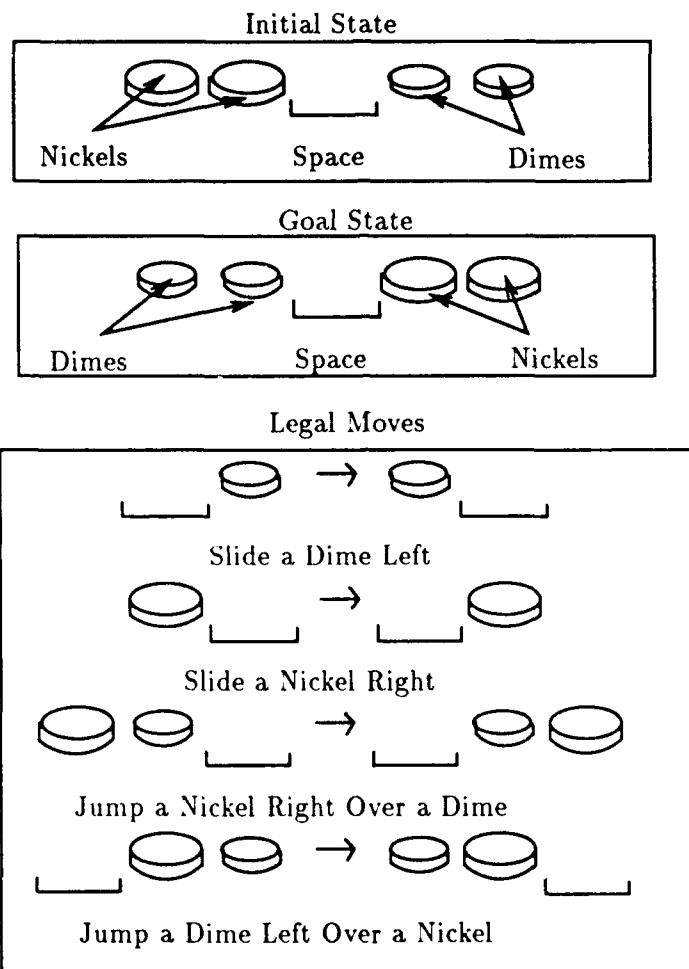


Figure 2.2. Graphical Representation of the Problem

```

(deffacts initial "defines initial and goal states"
  (sequence nickel nickel space dime dime)
  (goal dime dime space nickel nickel))

(defrule slide-a-dime-left "rule 1"
  (sequence $?x space dime $?y)
=>
  (assert (sequence $?x dime space $?y)))

(defrule slide-a-nickel-right "rule 2"
  (sequence $?x nickel space $?y)
=>
  (assert (sequence $?x space nickel $?y)))

(defrule jump-a-nickel-right-over-a-dime "rule 3"
  (sequence $?x nickel dime space $?y)
=>
  ((assert (sequence $?x space dime nickel $?y)))

(defrule jump-a-dime-left-over-a-nickel "rule 4"
  (sequence $?x space nickel dime $?y)
=>
  ((assert (sequence $?x dime nickel space $?y)))

(defrule found-a-solution "rule 5"
  (sequence $?x)
  (goal $?y)
  (test (eq ?x ?y))
=>
  (printout t "A solution has been found" crlf)))

```

Figure 2.3. The Dimes.clp Program

2.3.2 Expert System Example The second production system problem presents a simple rule based expert system to model the process used in optimizing the proper mix ingredients for concrete. The problem is necessarily simple in order to be presented easily. The following problem description is adapted from an example by Giarratano and Riley (32:328):

“Concrete consists of of cement, water, sand and coarse aggregate (such as small rocks) mixed in the proper proportions. The optimum proportions of the ingredients depends on the required strength of the concrete and the variability of the local materials. A mix is a trial amount of concrete which is made to determine if the proportions are correct for the desired application; a *slump test* is commonly used to test whether the proportions are correct. In the *slump test*, a trial batch of concrete is put into a cone and the cone is removed. The distance that the concrete mix spreads out after the cone is removed indicates the condition of the material. If the mix spreads out too much, it is “wet” indicating that sand and aggregate should be added. If the mix does not spread out enough, it is too “stiff” indicating that more water should be added. If the amount of spread is within these limits, then the mix is “workable” and can be used for the intended application.”

Again this is a relatively simple process, but it illustrates the utility of production systems for implementing rule-based expert systems. The problem description assumes an initial mix (specified as parts of the total mix), a specified minimum strength for the mix and an initial unknown slump of the mix.

The original problem described by Giarratano and Riley is probabilistic in nature, but this example ignores probabilistic factors for the sake of simplicity. Figure 2.4 shows the overall search process implemented in this rule-based expert system. As with the first problem, the program to solve this problem is written in CLIPS: The first rule calculates the strength of a proposed mix based on an arbitrary combination of the mix components. The next two rules take various actions depending on whether the proposed mix is strong enough; if the mix is not strong enough, cement must be added, if the mix is strong enough, a valid mix can be prepared. Rule number 4 also introduces the syntax used to retract a given fact from working memory: The CLIPS “<-” operator causes the number of a given fact to be assigned to a specific variable (such as ?old-action) so that the fact can be retracted “by name” when the rule is executed. Rule number 5 gets the results

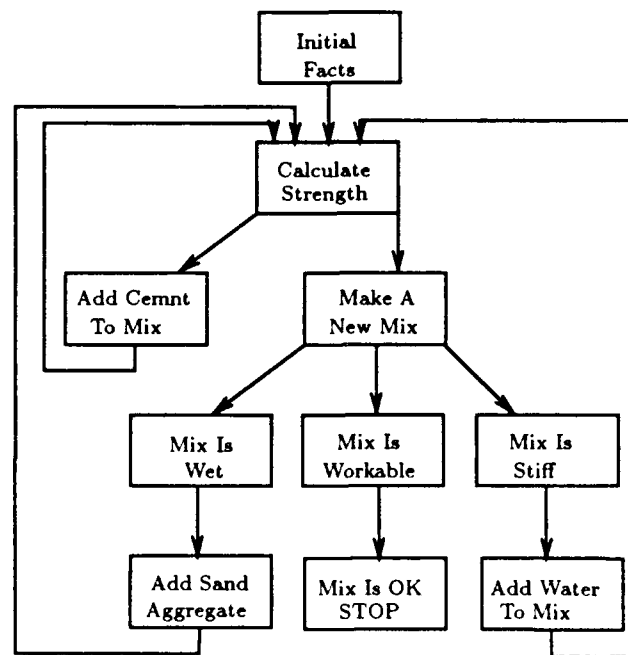


Figure 2.4. Hierarchy Chart for Concrete Master

("stiff", "workable" or "wet") of the slump test from the user. The results of the slump test then "triggers" three different possible courses of action. If the mix is not "workable" then either water or sand and aggregate mix is to be added to the next proposed mix. Once new proportions of a mix are proposed, the strength of the mix must be evaluated again and the cycle continues. If a "workable" mix is achieved, then a proper proportion of concrete components has been achieved.

2.4 Production Systems as a Search Process

Although they are often not abstractly considered as such, all production systems represent a search process. Examining production systems as a search process allows us to apply familiar principles toward better defining and understanding their behavior. At a very high level of abstraction, a production system, like a search process, starts with a defined problem and produces a solution (if there is one). This simple statement is sufficient for a high-level comparison, but significantly more detail is required to show how production systems actually work as a search process. The first step in describing

```

(deffacts initial-mix-and-goals
  (cement 4 parts) (water 2 parts)
  (sand-aggregate 3 parts) (slump value unknown)
  (goal strength 20000))

(defrule calculate-strength "rule 1"
  (cement ?a parts)
  (water ?b parts)
  (sand-aggregate ?c parts)
=>
  (bind ?x (/ (* (/ ?a ?c) 100000) (* ?b 2)))
  (assert (mix strength ?x)))

(defrule proposed-mix-is-strong-enough "rule 2"
  (mix strength ?x)
  (goal strength ?y)
  (test (>= ?x ?y))
=>
  (assert (action make-a-mix))
  (printout t "make a new mix"))

(defrule proposed-mix-not-strong-enough "rule 3"
  (mix strength ?x)
  (goal strength ?y)
  (test (< ?x ?y))
=>
  (assert (action add-cement)))

(defrule add-cement-to-mix "rule 4"
  ?old-action <- (action add-cement)
  ?old-amount <- (cement ?x parts)
=>
  (bind ?y (+ ?x 1))
  (retract ?old-action)
  (retract ?old-amount)
  (assert (cement ?y parts)))

(defrule remove-old-mix-result-get-new "rule 5"
  ?old-action <- (action make-a-mix)
  ?old-result <- (mix slump ?x)
=>
  (retract ?old-action)
  (retract ?old-result)
  (prinout t "Results of slump test?" crlf)
  (assert (mix slump ?y)))

(defrule add-water-if-mix-too-stiff "rule 6"
  ?old-amount <- (water ?x parts)
  (mix slump stiff)
=>
  (bind ?y (+ ?x 1))
  (retract ?old-amount)
  (assert (water ?y parts)))

(defrule add-sand-if-mix-too-wet "rule 7"
  ?old-amount <- (sand-aggregate ?x parts)
  (mix slump wet)
=>
  (bind ?y (+ ?x 1))
  (retract ?old-amount)
  (assert (sand-aggregate ?y parts)))

(defrule a-good-mix-has-been-found "rule 8"
  (mix slump workable)
  (cement ?x parts)
  (water ?y parts)
  (sand-aggregate ?z parts)
=>
  (printout t "A good mix is: " crlf)
  (printout t ?x "parts cement" crlf)
  (printout t ?y "parts water" crlf)
  (printout t ?z "parts sand-aggregate")
  (printout t crlf))

```

Figure 2.5. The "Concrete Master" Expert System Example

production systems as search processes requires a basic understanding of fundamental search concepts. It is then possible to tie the fundamental search concepts to the earlier description of production systems.

Typically a search is regarded as a sequential, centralized control strategy that accepts a problem in its initial state and then “searches” for a solution (64:55). Pearl describes a search process by its component parts (63:16):

1. Problem State
2. Operators
3. Control Strategy

A state is a representation (e.g. pictorial, textual, binary) of a problem at a given point between (and including) its definition and its solution. An operator is a method (or procedure) that can be applied to a state in order to create one or more new states. Finally, the control strategy decides which operators will be applied to which states in *which order*. There are a tremendous number of different control strategies, but they can be described by the following generic representation:

1. Start with an initial state.
2. Choose an available state to expand.
3. Apply some subset of the operators to the state to produce new state(s).
4. Evaluate the new states; if a goal state is found, keep it but do not attempt to expand it further.
5. Return to step 2

From a description of production systems and the general search process, it is relatively easy to show how production systems fit in the general search model. The components of the production system map directly to the components of the general search process:

- Working Memory \mapsto Problem State

- Rule Base \mapsto Operators
- Recognize-Act Cycle \mapsto Control Strategy

From the previous discussion of production systems, we know that Working Memory represents a problem state and that rules provide the means for modifying (operating on) this working memory. Figure 2.6 displays a graphical representation of the production system Recognize-Act cycle as a search process using the dimes and nickels generalized search example. During a given production cycle, only one state (denoted by contents of Working Memory) can be considered. Based on the contents of Working Memory, certain rules in the rule base become eligible to execute because all their “conditions” have been met. This is essentially the same action as applying operators to a chosen state to produce new states. The conflict resolution part of the production cycle corresponds to choosing a new path to expand in the general search process. Like greedy search methods, such as hill-climbing, production systems generally make an irreversible decision regarding which path to take, based on local information. In the production cycle, the actions of the chosen rule (operator) are now applied to the current contents of Working Memory (chosen state). Carrying out the action on the rule base corresponds to producing a new candidate state from the existing state, because the changes to Working Memory now make a new set of rules eligible to execute.

2.5 Time Complexity of Production Systems

If productions systems use greedy (i.e. typically with linear or polynomial time complexity) methods, why do they suffer from such poor performance? In order to determine the source of the poor performance, this section investigates the various phases of the production cycle and finally production systems as a whole. It turns out that each production cycle phase executes in polynomial time, but production systems as a whole constitute non-deterministic polynomial-complete (*NP-Complete*) problems! The *NP-Completeness* of production systems may be a little surprising, but tends to make sense when taking an abstract view of a production system problem implementation.

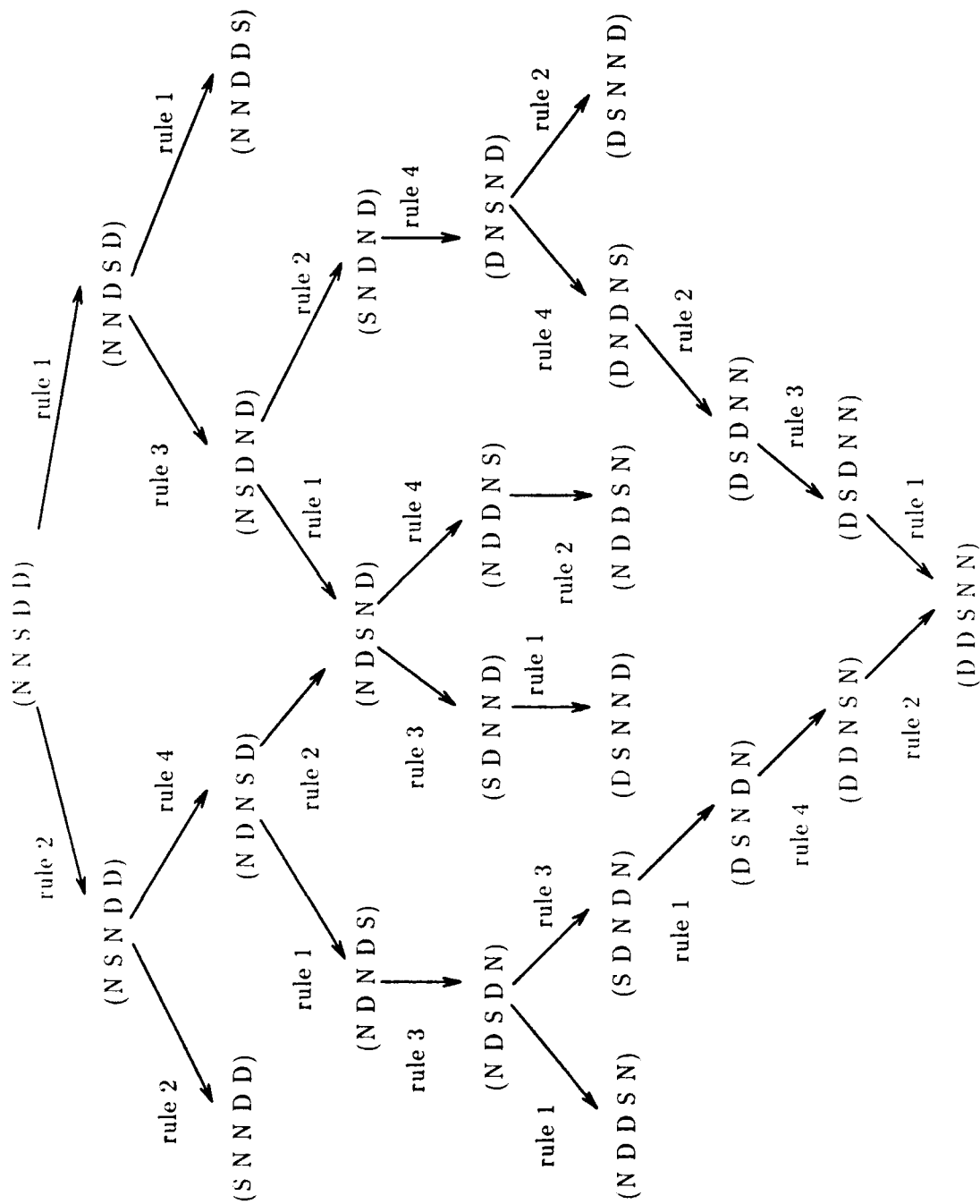


Figure 2.6. Production System Search Space

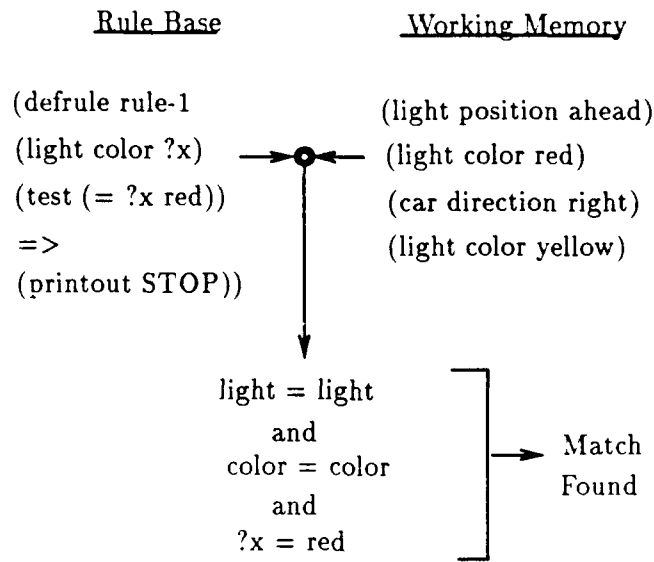


Figure 2.7. A Rule to WM Element Match

2.5.1 Match Phase Time Complexity The match phase is well known as the most compute intensive (in terms of time) phase of the production cycle (38:70) (36:12). A simple match algorithm compares the CEs (predicates) in the LHS of each rule in the rule base with the entire contents of Working Memory. To actually “match” a rule CE to a Working Memory fact, each variable or constant in the CE must be compared with each constant in a Working Memory fact. Figure 2.7 displays the “anatomy” of a CE to Working Memory fact comparison according to the following rules: (38:69)

- If the condition element is a constant, it will only match an identical constant.
- If the condition element is a variable, it will match any value.

In the example rule match, the variable ?x takes on (is instantiated with) the value red. Considering a large production system containing a number of rules and Working Memory facts, it is easy to envision the adverse effect of the match phase on execution efficiency.

A simple example illustrates the time inefficiency of the simple match process discussed in the previous paragraph. This simple match algorithm can be described by the following set of pseudo-code statements:

```

for all facts in working memory
  for all rules in the rule base
    for each CE in the rule LHS
      for each variable/constant
        CE variable/constant = working Memory constant?

```

For an r rule production system with an average of c condition elements per rule LHS, with w facts in working memory and all facts and CEs represented as object-attribute-value (OAV) triples, the time complexity of the match process is:

$$O(r \times w \times c \times 3)$$

A large production system with 1000 rules, 4 CEs per rule (average), 200 facts in Working Memory and using OAV triples requires 2.4×10^5 equality tests in the worst case. Forgy presents an equality test that requires approximately 4 machine instructions on the average (29:30). Given a modern 4 MIPS microprocessor, this match operation could require as long as 0.6 seconds!

2.5.2 Select Phase Time Complexity In practice, the select phase of the production cycle is not as "expensive" as the match phase, but possesses some significant complexities. There are a number of different rule selection strategies, but most modern production systems use one particularly compute intensive process. The concept of refraction means that a rule will never fire twice for the same set of variable instantiations. In the original dimes and nickels problem, this is why only one solution is indicated when two solutions actually exist; the rule that identifies that the goal state exists in Working Memory fires only once! The refraction process requires that the production system keep a list of all past variable bindings that resulted in previous rule firings. Therefore, every rule in the conflict set must be compared to the previous bindings for that rule that exist in the refraction list. Refraction is not the only part of the select phase, but it is the most computationally complex in terms of time, particularly as the number of production cycles executed increases. The refraction part of the select phase can be described by the following

pseudo-code fragment:

```
for all rules in the rule base
  for number of previous firings (specific rule)
    for each CE in the rule
      conflict set CE = refraction list CE?
```

For an r rule productions system, having executed n previous cycles (assumes each rule executed an even number of times), c CEs (represented as OAV triples) per rule LHS, the time complexity of the select process is:

$$O\left(\frac{r^2}{n} \times c \times 3\right)$$

For the previous production system example of 1000 rules and 4 CE per rule LHS, plus the additional consideration that 500 rules have fired so far, refraction alone requires 2.4×10^4 comparisons. This number of comparisons is significantly less than for match phase, but again this represents only part of the total select phase.

2.5.3 Act Phase Time Complexity The act phase is typically the least compute intensive phase of the match-select-act cycle. The number of actions performed (on working memory) as a result of a given rule firing is typically equal to the number of actions specified in the RHS of that rule fired. Assuming that a given action executes in constant time ($O(1)$), the order-of time complexity for the act phase is expressed as:

$$O(\max(A_n))$$

where A_n is the number of actions specified on the RHS of rule n . Thus, the rule with the largest number of RHS actions determines the order-of for them act phase. Measurements by Gupta and Forgy indicate that one RHS action per rule is the most prevalent case among production systems based applications (38:75). Conversely, there are examples of production systems (such as the SPAM high-level vision system currently under research by Carnegie-Mellon University) that have large numbers of actions for most rules (43:1).

2.5.4 Overall Time Complexity of Production Systems The previous parts of this section concentrate on the complexities of various phases of the production cycle instead of production systems as a whole. This examination was necessary in order to gain an appreciation for the computational complexities of production systems. However, these complexities have little to do with the overall analysis. Calculating the overall complexity of production systems is very difficult because of the non-determinism of its control structure. The following section argues that forward-chaining production systems are *NP-Complete*. Bassard and Brately present the following definition of *NP-completeness* (10:325):

A problem X is *NP-Complete* iff

- (i) $X \in NP$
- (ii) $X \leq_p^t Z, NP-Complete(Z)$

This means that: 1. a problem X has exponential time complexity and 2. that an X type problem can be transformed into a known *NP-Complete* Z type problem in polynomial time (\leq_p^t). For example, proving a given problem is *NP-Complete* involves 1. showing that it cannot be solved in less than exponential time ($O(c^n)$) and 2. describing how that problem can be mapped into a known *NP-Complete* problem such as the Traveling Salesman Problem (TSP).

O'Reilly and Cromarty present production systems as an exponential order-of problem through a number of simplifying assumptions: (61:253)

- Assume only conjunctions (ands) within the LHS and RHS of each production rule.
- Assume a uniform number of rules become eligible as a result of a given rule firing.
- Assume the cost of matching variables between LHS CEs can be ignored.
- Assume the depth of a given rule (e.g. if two rules share an LHS CE, then either both or neither of the RHS of those rules will result in the firing of additional rules.) is uniform across the rule base.

It is difficult to visualize the effect of the fourth and final assumption unless the production system is thought of in "levels of reasoning". That is, the reasoning performed by one level

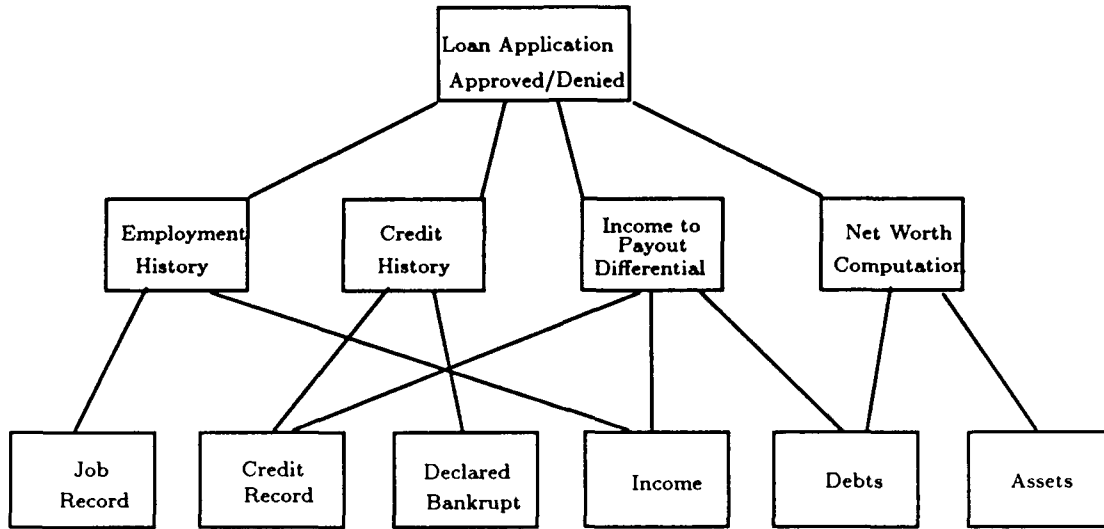


Figure 2.8. Levels of Reasoning for a Simple Expert System

of the system supports the reasoning accomplished by a higher level which in turn supports reasoning accomplished at an even higher level. Figure 2.8 gives a visual example of how these “levels of reasoning” are incorporated in expert systems for a personal loan advisor.

Considering the assumptions made previously, it is relatively straightforward process to show production systems possess exponential time complexity. Given that each rule’s antecedent contains k consequents, of which $l < k$ are true, (i.e. “ l of the k consequents trigger rules for which the antecedents, or antecedent-conjuncts, are now true”) and f as the “ultimate” (finite) “depth” of the rule base. The number of rules that will fire is given by (61:253):

$$\sum_{i=1}^f kl^{i-1}$$

which is linear in k and l in exponential in f . O’Reilly and Cromarty point out that a “flat” rule base ($f = 1$) is not very knowledgeable, thereby making most production systems exponential in time (61:253). Given a non-deterministic Turing Machine with $l \times f$ processors, this same problem can be executed in (pseudo) polynomial time. This satisfies the first requirement for showing production systems are *NP-Complete*.

The second part of proving the *NP-Completeness* of production systems involves transforming a known *NP-Complete* problem to a production system problem in polynomial time. Recall that Post originally developed production systems as a computational model and proved that they are equivalent to a Turing Machine. This means that any *NP-Complete* problem can be represented using the production system paradigm. In his thesis, Shakley presents a straight forward transformation of a well known *NP-Complete* problem, the Set-Covering-Problem, to a production system. The transformation is as follows (70:86):

- The sets to be included make up a C vector. If they were included, then the component of the C vector would be 1, otherwise it would be 0.
- The F vector is the union of all the sets with a 1 in the C vector.

This transformation can be accomplished in polynomial time, therefore meeting the second requirement for proving *NP-Completeness*!

2.6 Fast Matching Using Rete

The previous section's analysis shows the match phase to be the most computationally "expensive" part of the match-select-act cycle. Past research indicates that the match phase accounts for more than 90% of a typical production system's execution time, even when using specialized algorithms (35:4) (38:70). Consequently, much research directed at increasing the execution speed of production systems centers on improving the time efficiency of the match phase. The Rete match algorithm is one of the most time efficient match algorithms developed to date. Rete takes advantage of typical production system attributes to provide dramatically increased performance for most production system based applications. The Rete algorithm has been so successful that it has been incorporated into a number of commercial production system tools, such as NASA'S CLIPS and Inference Corp's Automated Reasoning Tool (ART).

2.6.1 A High Level Overview The Rete algorithm achieves significant increases in execution speed by exploiting two prevalent characteristics of production systems (36:15):

- First, in a typical production system application, only a very small fraction (typically 0.5% (36:16)) of the Working Memory actually changes during a given match-select-act cycle.
- Second, in a typical production system, similar CEs often appear repeatedly on the LHS of rules in the rule base.

Rete takes advantage of the small Working Memory change by storing CEs satisfied during previous production cycle; it can then use them to help satisfy rules during subsequent production cycles. This “state-saving” feature means that only recent changes to Working Memory need to be evaluated during the current match phase instead of evaluating the entire Working Memory. Rete exploits the commonality of CEs in the “if” (LHS) of rules in the rule base by effectively eliminating redundant CEs. In eliminating redundant CEs, Rete evaluates a given CE only once each cycle, even though this CE may exist in the LHS of many different rules. Eliminating redundant CEs tends to significantly reduce the number of pattern matching tests that need to be accomplished during each production cycle (42:2).

2.6.2 Details of the Rete Algorithm The Rete algorithm uses a special type of discrimination network compiled from the LHS of rules in the rule base to perform efficient pattern matching. Figure 2.9 contains an example Rete network compiled from the two following simple production rules:

```
(defrule cleared-to-land
  (aircraft (name ?call-sign)
    (assigned-to ?runway-num)
    (cleared-for approach))
  (runway (name ?runway-num)
    (status clear))
=>
  (modify (aircraft (cleared-for landing))))

(defrule cleared-for-approach
  (aircraft (name ?call-sign)
    (assigned-to ?approach-path)
    (cleared-for holding))
  (approach-path (name ?runway-num)
    (status clear))
  (navaids (for ?approach-path)
    (status nominal))
=>
  (modify (aircraft (cleared-for approach))))
```

The Rete network is actually generated at compile time, before the production system application is actually executed. At run time, entities (i.e. tokens) containing a flag, a

list of time tags and a list of variable bindings flow through the Rete network during each match phase. At each level in the Rete network, two tokens are joined to form a new token if and only if all of the variable bindings between the original two tokens are consistent. When a new token binding is no longer possible, the token is retained inside the network until a later production cycle when another match will be attempted. A token reaching the bottom of the network contains a complete and valid list of variable bindings associated with a rule that is now eligible for execution. The following discussion contains additional details on how the Rete algorithm constructs the matching network and the various nodes used in the network.

When a set of rules is first loaded into a Rete shell, a network compiler parses and translates the rules into a Rete network. However, instead of generating executable code, the compiler actually creates a data structure that the inference engine uses to improve the efficiency of the match phase (41:3-4). For each CE in the LHS of different production rules, the network compiler creates and links together nodes to test for consistency of both constant and variable bindings. When identical CEs occur in the LHS of different rules, the network compiler creates a single shared test node. To avoid performing the same tests done during previous match phases, the network compiler also creates memory nodes to retain the current state of working memory. Finally terminal nodes are added to the "bottom" of the Rete network to hold variable instantiations of a candidate rule that becomes part of the conflict set (of rules eligible to execute). The specific functions of each of the Rete data tokens and the Rete network nodes are now discussed in detail based on Forgy's original work (36:12).

- *Data Token:* A data token consists of a tag and a list of data elements. The tag indicates whether the items in the list of data elements are to be added to or removed from WM. A "+" tag means "add this list of elements", a "-" tag means "remove this list of elements". The list of data elements correspond to instantiated CEs from the LHS of rules in the rule base. From the example in Figure 11, two different token examples are:

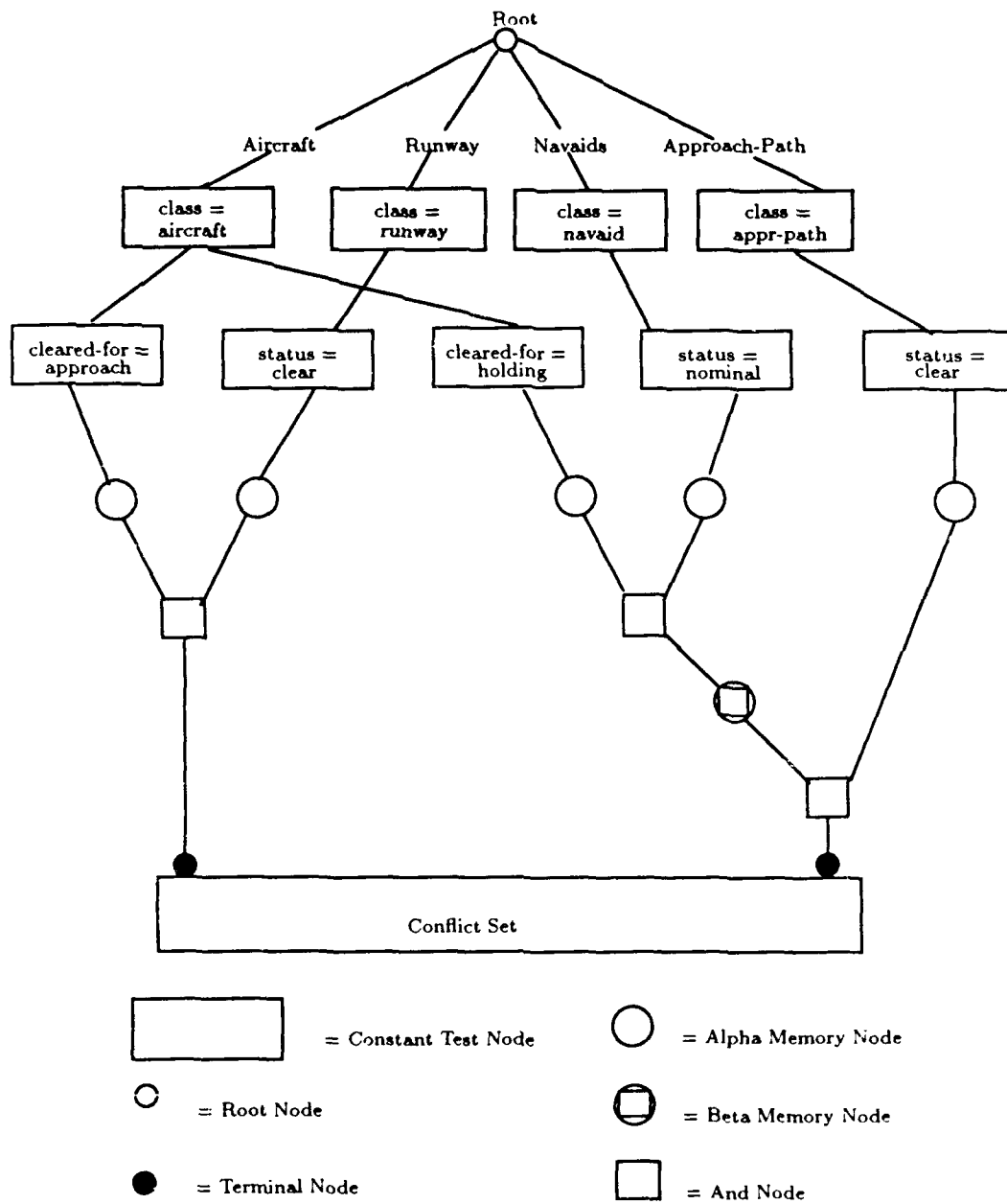


Figure 2.9. An Example Two Rule Rete Network

(+ (aircraft name United223 assigned-to 14L cleared-for approach))
(+ (runway name 14L status clear))

- *Root Node:* The root node represents the “top” of the Rete network. The root passes copies of tokens (corresponding to changes to WM) to all the successor (constant test) nodes in the network.
- *Constant Test Nodes:* These nodes are responsible for performing intra-condition testing of the tokens sent from the root node. All constant fields in the token must match the specified constants in the constant test node in order for the token to be propagated further. Each constant test node is actually a series of test nodes that check for the equality of one field only. For the data token example, one part of the constant test node ensures that the first field of the token is equal to “aircraft”. Using this test, the first token is propagated, but the second token is not. If the same variable appears twice in the same data token, constant test nodes also check to ensure that the possible variable bindings are consistent.
- *Alpha Memory Nodes:* When a token successfully passes all intra-condition tests done by the constant test nodes, the token is said to *partially match* the corresponding condition element. If the token’s tag is a “+” it is added to the contents of the memory node and a copy of the token is propagated to the next node(s) in the network. If the token’s tag is a “-”, then the corresponding “+” token (assumed to already exist in the memory node) is deleted and a copy of the “-” token is propagated to the next node(s) in the network. In the continuing case of the previous example, this means that the “aircraft” token is retained in the alpha memory node immediately following the constant test nodes, as well as being propagated to the next node(s) in the network.
- *Beta Memory Nodes:* Like alpha memory nodes, beta memory nodes store partial CE matches, but alpha memory nodes store only matches for individual CEs whereas beta memory nodes store partial matches for two or more CEs. Beta memory nodes

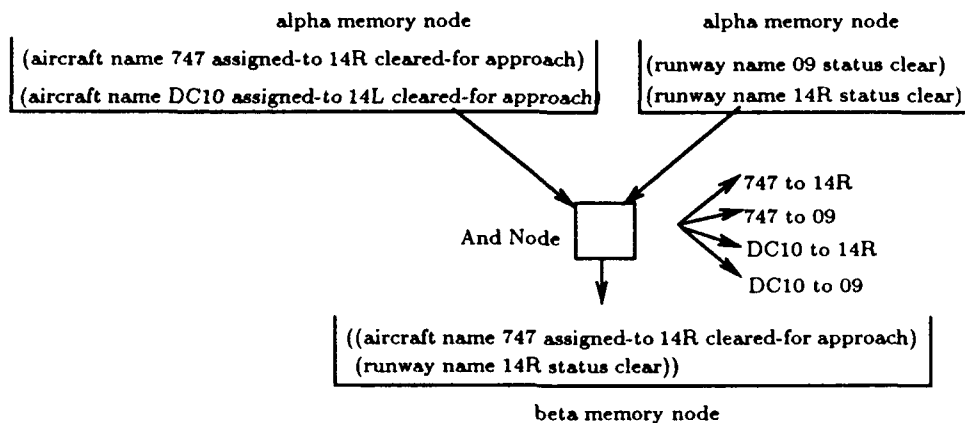


Figure 2.10. Example of an *and* Node Activation

respond in an identical manner to alpha memory nodes upon that arrival of “+” or “-” tagged tokens.

- *And Nodes:* The primary function of this type of node is to test for inter-condition consistency of variable bindings between partially matched tokens that it receives from two memory nodes. The *and* node forms a *cross product* of the tokens presented to it and passes on a *join* of these tokens **only** if the variable bindings between the two inputs are consistent. Figure 2.10 shows in detail how this process takes place using an extension of the continuing example. One input of an *and* node must come from an *alpha memory* node while the other input can come from either an *alpha memory* node or a *beta memory* node. The output of an *and* node goes to either a *beta memory* node or a terminal node.
- *Not Nodes:* Not nodes are used to test for the absence of a particular CE in WM (a feature available in OPS-5, but not available in some other production system languages). As with *and* nodes, *not* nodes receive input from two memory nodes (one of which must be an *alpha memory* node) and attempt to form valid cross-product joins between them. The major difference between a *not* node and an *and* node is that a joined token is propagated only when a match is *not* made.
- *Production Nodes:* Production nodes represent the terminal nodes in a Rete network. There is one terminal node corresponding to each rule in the rule base. Whenever a

token with a "+" tag reaches the one of these terminal nodes, the rule corresponding to this node (with variable instantiations taken from the token) is entered into the conflict set. If a token with a "-" sign reaches a terminal node, the corresponding rule (with a matching set of variable instantiations) is removed from the conflict set.

2.7 Why Rete is Better

Using the Rete network, only the CEs affected by the previous production cycle's act phase need to be updated as opposed to iterating the match over all elements in WM. The two different phases of the Rete network update are: the *selection* phase and the *state-update* phase. During the *selection* phase, all of the constant test nodes are updated based on WM changes from the most recent rule execution. The *state-update* phase consists of evaluating all of the other types of nodes in the Rete network: *alpha memory*, *beta memory*, *and*, *not* and *production*. The *selection* phase typically requires only 5% to 25% of the total Rete network update time whereas the *state-update* phase takes between 75% and 95% of the total time. Other important observations based on the characteristics of a number of different production system based applications can be found in (38) and (37).

The Rete algorithm's state-saving approach avoids tremendous computational overheads due to redundant computations, however this state-saving feature has a price of its own. Although the Rete network concepts are fairly easy to visualize, efficiently implementing them is another matter entirely. Source code for different versions of the Rete algorithm in LISP (OPS-5) C (CLIPS and OPS-83), and Ada (Ada-CLIPS) easily attest to the implementation complexity. This complexity translates to a greater number of machine instructions for each match action when compared with the (simple) exhaustive match method. Extensive research by Gupta shows that each working memory action using Rete requires approximately 1800 machine instructions as opposed to 1100 machine instructions for the exhaustive match algorithm; additionally, the Rete algorithm must only process the number of WM changes specified by a single rule ($(i + d) = (\text{inserts} + \text{deletes})$) instead attempting to match the number of elements in WM (denoted by s) (36:16). Thus, in order for the exhaustive match algorithm to compare favorably with

Rete:

$$1000 \cdot s \leq 1800 \cdot (i + d).$$

Observing that $(i + d)/s = 0.61$ indicates that at least 61% of WM (on the average) must change during each production cycle in order for exhaustive match algorithm to overcome the advantage of the Rete algorithm (36:17). Again, previous research shows that, with a very few exceptions, most production systems applications change much less than 61% of WM each cycle (38:30).

2.8 Conclusion

Production systems represent a complex search process whether they are being used to represent cognitive thinking or to solve generalized search problems. As a result of their flexibility and expressive power, production systems are also very compute intensive. The process of looking for the next possible states in the search process (the match phase) tends to be very costly in terms of time. Even using specialized algorithms, such as the Rete match algorithm, the match phase requires significantly more time than any other phase of the production system execution. Additionally, the Rete algorithm cannot change the overall exponential complexity of production systems because it does not address the “levels of reasoning” argument. Although the Rete algorithm can increase the execution speed of most production systems, production systems are still *NP-Complete*. Therefore, production systems will always possess the special difficulties associated with problems in the *NP-Complete* class. The most notable of these problems is that the time required to obtain a solution increases exponentially as the “size” (actually complexity) of the expert system increases. This exponential complexity will limit the size and complexity of production system based applications that can be efficiently solved using current techniques.

III. Parallel Computing Concepts, Problems and Architectures

3.1 Introduction

Over the years, technological advances have dramatically increased the performance of sequential computer architectures; however, the laws of physics will ultimately limit the achievable performance of these architectures (30:2). Parallel computer architectures offer a complementary approach to sequential architectures in solving problems of ever increasing complexity. The main thrust behind parallel computing is the explicit use of multiple processing elements (often computers in themselves), to solve complex problems many times faster than on a comparable sequential architecture¹. Many computer scientists foresee standard computer designs of the future employing massive parallelism; that is, computers containing an arbitrarily large number of processing elements. As attractive as parallel computing sounds, there are still a great number of issues that must be successfully addressed before massively parallel computers can be used to solve complex problems efficiently. This chapter explores both the concepts of parallel program design and the architectures currently available for parallel processing in an effort to give the reader some feel for the existing problems inherent in parallel computing.

3.2 Essential Concerns in Parallel Computing

The essential concerns in designing programs for parallel computer architectures generally fall into two categories: correctness and performance². Clearly a program design must be correct in order to guarantee solution of the problem; however, ensuring the correctness of parallel program designs is generally more difficult than for sequential designs due to the existence of multiple threads of control. In addition to correctness, the parallel program design should make the most effective use of the computational resources of the available parallel computer architecture; this means not only designing the program for correctness, but for time and space efficiency as well. Recent advances in semiconductor

¹Comparable is used to indicate that the single processor of the sequential architecture is assumed to possess the same computational "power" as each processing element of the parallel architecture

²The concept of performance is used to denote efficiency (in terms of both time and space) of a parallel program design; this reserves the use of "*efficiency*" for a later definition in order to avoid confusion

memory and microprocessor technology have reduced many of the past concerns relating to space efficiency, therefore more emphasis is placed on time efficiency when attempting to determine parallel program design performance. Although parallel architectures constitute a potentially tremendous increase in computational power, difficulties with either correctness or performance can vitiate their usefulness. The following section discusses both of the essential concerns (correctness and performance) of parallel program design and how they relate to the development of "successful" applications for parallel computer architectures.

3.2.1 Design and Development of Correct Parallel Programs In the quest for improving performance through parallel architectures, the concept of program correctness is often left out entirely. The majority of the literature on programming parallel computers never considers this critical aspect of program design and development. This omission of program correctness concepts is potentially dangerous when dealing with the complexities of parallel program design. Developing programs for parallel (computer) architectures is nearly always a more difficult task than developing programs for sequential architectures (15:5). The existence of multiple processing elements compounds the task of isolating and correcting parallel program mistakes significantly. As a direct result, parallel program correctness and careful development methods are crucial to the successful development of an implementation. Incorrect parallel program design may yield implementations that encounter problems related to: deadlock, starvation, termination detection and race conditions. Any of these problems may cause a computer to halt execution abnormally (terms such as "hang" or "crash" are appropriate), produce incorrect answers or both (56:593). A correct program on the other hand, always produces the correct results (as long as system problems are not encountered) in a predictable and repeatable manner.

The UNITY³ approach formalized by J. Mani Chandy and Jaydev Misra provides a "foundation for parallel program design"(15:4). The UNITY approach isolates the program designer from the specifics of a given parallel architecture in the same way that modern compilers isolate programmers from the machine specific characteristics. The UNITY

³Unbounded Non-deterministic Iterative Transform

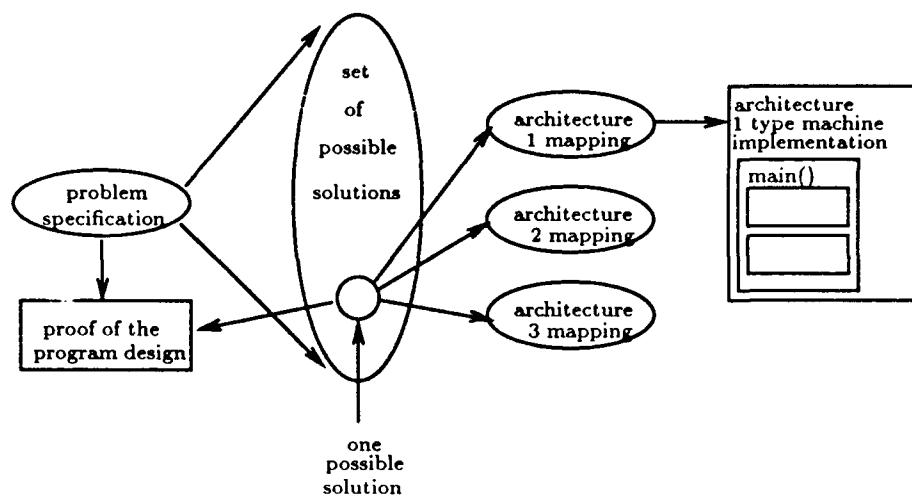


Figure 3.1. Overview of the UNITY Approach

approach allows the program developer to postpone architecture dependent decisions until later in the design process because the initial UNITY program is typically architecture independent. Independence from the specific parallel architecture is achieved through the use of temporal predicate logic combined with a simple, but powerful state transition model (15:9). The power of the UNITY approach is that a proof of program correctness can be extracted from the high-level specification of the program design using a combination of standard logic operators (reductions) and special temporal logic operators. Proving that the high level design is correct provides an essential framework for developing a specific implementation that is correct. The UNITY approach also provides a method for transforming high level program designs into intermediate forms using correctness-preserving *mappings*. These intermediate forms maintain the correctness of the original UNITY program, but provide a basic structure for developing implementations on specific parallel architectures. At this point, source code for the executable program can be written based loosely on the intermediate form.

Figure 3.1 provides an overview of a parallel program development process that utilizes the UNITY approach. Other methods for parallel program development are available: Communicating Sequential Processes (CSP) by C. A. R. Hoare (45), the Parallation model defined by Gary Sabot (66), and the Actors model formalized by Gul Agha (2). However,

none of these methods possess all three of the important concepts expressed by the UNITY approach: high-level, explicit expression of parallelism; extrication of proofs from the basic program design; and mappings of the initial design to various parallel architectures while maintaining correctness. Parallel program correctness is of critical importance, thus the UNITY syntax is used in all parallel program examples in this document. However, the main goal is not in proving parallel program correctness, therefore it takes second precedence to the concept of parallel program performance. Appendix A contains a general description of the UNITY approach and provides a simple example to illustrate its use.

3.2.2 Parallel Program Performance Unfortunately, a correct program design does not guarantee that the results will be produced in an efficient manner when the actual design is implemented on a specific computer architecture. An efficient parallel program design enables the maximum use of all processing elements of the computer architecture toward solution of the problem. This in turn implies that an efficient parallel program design will run many times faster on a parallel architecture than on a comparable sequential computer architecture. There are a number of significant obstacles related to developing efficient parallel program designs related to the problem characteristics, the program design methodology used and the parallel architecture. Stone describes how these obstacles manifest themselves as the following overhead costs (73:280):

- Delays introduced by communicating data between processing elements in the parallel architecture, also known as *communication overhead*.
- *Synchronization delays* introduced when one processor must wait for another to complete a given task before it can proceed.
- Lost efficiency when one or more processors runs out of useful work (tasks) and becomes idle, a condition known as *load imbalance*.
- Efficiency lost through computations not required for a sequential implementation, composed of:
 - Time costs associated with the use of non-optimal (time) sequential algorithms and data structures.

- Time costs associated with control functions such as task ordering and task scheduling.
- Time costs associated with computations not required by equivalent sequential implementations.

A parallel program design is either correct or incorrect, but parallel program performance is not so "black and white". Because performance is not as definite, some means of quantifying a given design's (and its subsequent implementation's) performance is required. Briefly presented are some of the more fundamental measures of parallel program performance: parallel program speed-up (S) and parallel program efficiency⁴ (ϵ). Parallel program speed-up determines how much faster a given problem can be solved on a specific parallel architecture than on a comparable sequential architecture. The measure of parallel program efficiency relies on the speed-up metric, but is significantly different because it provides a measure of how efficiently the parallel program design uses the processing elements of the parallel architecture in solving the problem (**not** how much faster the design is). To better grasp these measures, specific formulas for each of these metrics need to be introduced.

Speed-up is one of the most important and widely used measures of parallel program performance. The speed-up factor (S) is expressed as the ratio of time required to solve a particular problem on a parallel architecture ($T_{parallel}$) compared with the time required to solve the same problem on a comparable sequential architecture ($T_{sequential}$)⁵ (30:55). The speed-up factor can be expressed in equation form as:

$$S = \frac{T_{sequential}}{T_{parallel}}$$

where $T_{parallel}$ is a combination of the actual time required to compute a solution to the problem using a number of processing elements plus a combination of the overhead costs

⁴This metric is different than the measure of time and space complexity of a sequential or parallel program (algorithm) design with the same name

⁵The additional caveat of optimal algorithm use is often included in the definition of speed-up

discussed previously. Therefore $T_{parallel}$ can be written as:

$$T_{parallel} = \frac{\overbrace{T_{sequential}}^{computation}}{N} + \overbrace{T_{comm} + T_{synch} + T_{imbalance} + T_{comp}}^{overhead}$$

The above equation assumes that all overhead costs can be independently quantified and that the problem can be divided into N independently executable tasks. Although the equation provides a starting point for a discussion of parallel program performance, later examples show that neither of the above assumptions hold for all problems. The definition of parallel program efficiency follows directly from the definition of speed-up; efficiency is defined as the speed-up achieved per node. Efficiency can be written in equation form as:

$$\epsilon = \frac{S_N}{N}$$

where S_N is the speed-up achieved using N processing elements.

Using the definitions for speed-up and efficiency, it is possible to describe the desired performance of any parallel program implementation even before looking at any specific problems! One of the goals in using multiple processors for solving any problem is to reduce the amount of time required to solve the problem compared to sequential architectures. When $T_{parallel}$ is less than $T_{sequential}$ speed-ups greater than one (positive) result. Ideally, we would like a given parallel architecture with N processors to solve every problem N times faster than a comparable sequential architecture. This level of speed-up is known as *linear speed-up* and results in "perfect" efficiency of 1. Linear speed-up also implies that the overhead factors in the equation describing $T_{parallel}$ are essentially 0. Eliminating all of these overhead factors is a very difficult proposition for most problems and impossible for many.

Minimizing the overhead costs and dividing (or balancing) the problem evenly between available processors is dependent on a number of different factors. Generally these factors (30:3) fall into one of the three following areas:

- Characteristics of the problem to be solved.

- The program design and decomposition techniques employed.
- The chosen parallel architecture and its characteristics.

The following section concentrates on the different parallel program design techniques that can be employed for different types of problems. Section 3.4 then discusses the characteristics of different parallel computer architectures and how these characteristics relate to their ability to solve different type of problems. Finally, the last section briefly summarizes the concepts of problems and architectures and melds them into a common framework.

3.3 Decomposing a Problem/Composing a Design

As with the problem solving process on sequential architectures, algorithms are typically the starting point for any parallel program design. For many problem types, highly efficient sequential algorithms already exist; but in many cases, the problems have simply become too computationally complex to be efficiently solved on standard sequential architectures (30:26). Oftentimes, the optimal sequential algorithm can be easily extended to support a parallel design. However, if the optimal algorithm represents an inherently sequential process or is ultimately incompatible with the target architecture, the designer must consider the alternatives of adapting non-optimal algorithms or developing entirely new parallel algorithms. The choice of a specific approach to solving the problem significantly affects and often limits many later choices; however, this selection is a necessary first step in the parallel program design process.

An integral part of a parallel program design describes how the high-level program (algorithm) is broken down into independently executable tasks, a process also known as *decomposition*. Noting that a program is composed of both control structure and data structure (79), it makes sense that a parallel program decomposition can be terms of data structure, control structure or both (4:4-1). Most highly parallel programs appear to use data decomposition techniques and in most applications, true functional (or control decomposition) techniques appear to yield only small amounts of effective parallelism (31:900).

However, for the sake of completeness, both data decomposition and control decomposition techniques are presented.

At this point, the UNITY approach again becomes important in the parallel program design process because it helps to formally define the inherent parallelism in a specific algorithm and to implicitly define a number of decomposition options. The UNITY approach allows any high level program implementation to be written as a series of assignment statements capable of being executed in parallel. Using the UNITY syntax, a parallel program designer can expose all sources of parallelism in a specific problem solving approach (algorithm). This allows a detailed examination of the complexity of each independently executable part of the program and aids in making intelligent decisions concerning explicit decomposition of the program (design). Using the formal mapping techniques provided by UNITY, an explicit decomposition based on the chosen parallel architecture can be selected at a later point in time. In the decomposition approach for one parallel architecture, assignment statements can be assigned to separate processing elements. Therefore, a UNITY program containing M assignment statements can implicitly define $N \times M$ possible decomposition options for an architecture with N processing elements.

Problem decomposition introduces three important problem characteristics that later become major issues when attempting to design efficient programs for specific parallel architectures:

- problem granularity
- problem connectivity
- problem homogeneity.

Granularity refers to the relative size of independently executable tasks to be completed in solving a given problem. The relative size of tasks can be measured in a number of different ways such as worst case order-of, average order-of or actual profiler line execution counts if very accurate time measurements are required. Problem connectivity describes the interdependencies of the tasks that make up the problem both in terms of how many tasks affect a given task and how often these tasks affect the given task. Problem granularity

and problem connectivity are roughly analogous as large grain problems typically have very low connectivity and small grain problems typically have very high connectivity. Problem homogeneity is simply the variation in the size of the tasks to be completed in solving the problem. Homogeneity is a much more abstract problem characteristic than problem granularity as it is difficult to make a direct measurement of problem homogeneity. Therefore, three categories that define the spectrum of general problem characteristics typically encountered in designing parallel programs are (31:903):

- *Synchronous* - Problems that have a known number of roughly even sized tasks requiring global synchronization at various points during execution.
- *Loosely Synchronous* - Problems having some roughly defined number of tasks with approximately predictable sizes requiring infrequent local and/or global synchronization.
- *Asynchronous* - Problems that have a very approximate number of unevenly sized tasks with dynamically determined (determined during execution) synchronization requirements.

The following section on data decomposition provides a problem example that falls roughly into each of the first two categories. Techniques for solving problems in the third category are discussed, but a detailed example is beyond the scope of this chapter.

3.3.1 Data Decomposition Techniques and Examples The data (or domain) decomposition technique is characterized by large sets of data that can be divided into a number of smaller sets of data, each of which can be operated on in parallel. A number of processing elements can then apply an essentially identical control structure (algorithm) to each of these data subsets independently. A problem is *perfectly parallel* if the final problem solution can be computed independently by a number of independent processors based only on the original input data. This means that no communication between processors and no redundant computation is required during program execution. Large-sized perfectly parallel problems can be effectively and efficiently implemented on a large variety of parallel architectures (31:905). In a survey of parallel application areas, Fox notes that

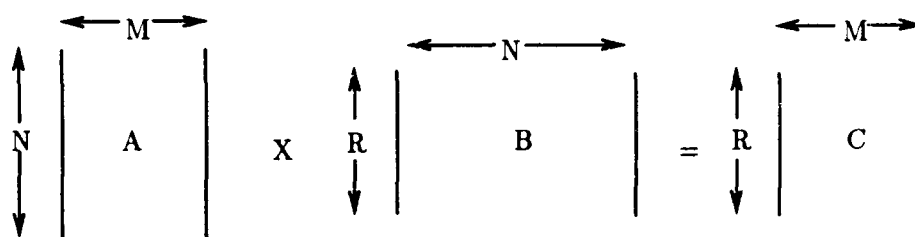


Figure 3.2. Matrix Multiplication Example

these “embarrassingly parallel” problems constitute only a small portion of all problems studied (31:927). Typically, a problem under investigation introduces one or more of the parallel overhead factors discussed earlier. The following examples provide an illustration of how each of these overhead factors can directly result from a given problem.

3.3.1.1 Parallel Matrix Multiplication The basic data (domain) decomposition technique can be easily illustrated using the generic matrix multiplication example shown in Figure 3.2. The standard sequential matrix multiplication algorithm has many independent parts that can be effectively represented by the UNITY program in Figure 3.3. The matrix multiply UNITY program is very similar to an abstract description of a sequential matrix multiplication except that the two outer loops of the sequential algorithm have been replaced by the \parallel operator which specifies synchronous parallel (all at once) operations. Of course, the sum of N multiplication products cannot be computed “all at once”, but using N processing elements, the sum can be computed in $O(\lceil \log_2 N \rceil)$ time instead of the $O(N)$ time required by the sequential method. Now that the parallelism inherent in the program design is completely exposed, it is possible to look at several different decomposition approaches.

This problem provides an excellent example of how the scale of parallelism in a specific problem changes with the problem decomposition. At the highest level of decomposition, there is one matrix multiply problem with complexity of $O(N \times M \times R)$. Decomposing the problem by solution matrix row yields M parallel problems with $O(N \times R)$ complexity. Similarly, decomposing the problem by solution matrix column gives R parallel problems with $O(N \times M)$ complexity. Reducing the problem to calculate each element of the matrix

Program Matrix

```

initially /* Define A and B matrices of numbers, C as all zeros */
A[0..M - 1, 0..N - 1] B[0..N - 1, 0..R - 1] C[0..M - 1, 0..R - 1]
⟨||i : 0 ≤ i ≤ M - 1 ::
    ⟨||k : 0 ≤ j ≤ N - 1 ::
        C[i, j] = 0⟩⟩

assign
/* for all rows in the solution matrix C */
⟨||i : 0 ≤ i ≤ M - 1 ::
    /* for all columns in a given row of the solution matrix c */
    ⟨||k : 0 ≤ j ≤ R - 1 ::
        /* each C element is a synchronous add of products */
        C[i, k] := ⟨+j : 0 ≤ j ≤ N :: A[i, j] × B[j, k]⟩
    ⟩
⟩
end {Matrix}

```

Figure 3.3. Matrix Multiplication UNITY Program

in parallel results in $M \times R$ parallel problems with $O(N)$ complexity. Note that for the first four decompositions, none of the tasks depend upon intermediate results from other tasks and is therefore perfectly parallel. Finally the problem can be reduced to the smallest operation that can be performed in parallel: computing the product of two matrix elements then adding these products together using the binary tree method. At this lowest level of parallelism there are $N \times M \times R$ problems, however the tasks are no longer the same size!

Considering the tasks to solve a simple 2×2 matrix as a time-space lattice as shown in Figure 3.4, it is relatively easy to see that the the binary add of the individual products results in a decreasing number of tasks at each "level". The largest size task for this type of decomposition will have complexity $O(\log_2(M \times R) + 1)$ or $O(3)$ in the 2×2 example problem. At higher levels of parallelism, this particular problem is completely homogeneous and disconnected, but at the lowest level of parallelism the task sizes are no longer homogeneous and connectivity between tasks are introduced. If each task in the low level parallelism model was assigned to a different processor, some processors would finish before others. In this scenario, parallel speed-up and efficiency are constrained by

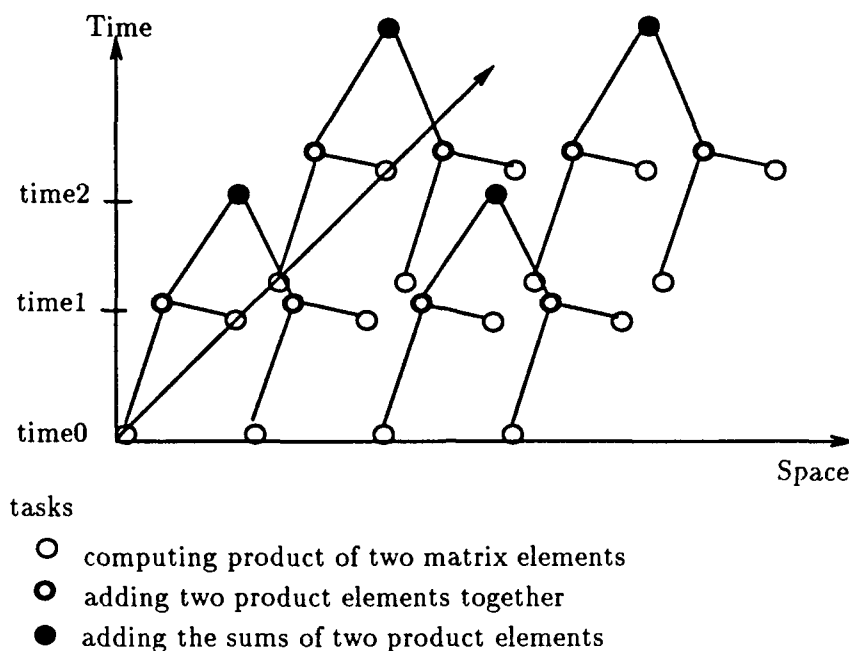


Figure 3.4. An Example Matrix Multiplication Time/Space Lattice

the task(s) that take the longest to complete. Processors with task sizes less than the largest task size run out of tasks and the condition of load imbalance results. Additionally, the binary add phase requires that intermediate sums be passed to tasks at the next level thereby adding communication overhead not present in the higher level decompositions.

Despite the potential irregularities involved in mapping the problem domain to a specific parallel architecture (for instance mapping a 5×5 matrix to 4 processors), Homogeneous problems, like matrix multiplication, do not pose any fundamental difficulties when using data decomposition (30:7). As shown in the UNITY program of Figure 3.3, the size and number of tasks at any given level of decomposition is known a priori (i.e. prior to execution time). As a direct result of knowing this information, an efficient method for the partitioning of tasks to processing elements can be specified at compile time. In fact, a number of commercially available vectorizing compilers are capable of automatically producing efficient parallel code for homogeneous vector-oriented problems from sequential program code (6:19). The fixed assignment of tasks to processing elements is known as *static load balancing*. A recent survey of parallel program applications indicates that

approximately 40% of all problems fall into the realm of homogeneous problems (31:927). This leaves a significant number of problems that fall into the other two areas: loosely synchronous type problems and asynchronous type problems.

3.3.1.2 The Parallel N-queens Search Problem Solving the parallel N-queens search problem provides a good example of a non-homogeneous type of problem and the alternate data decomposition techniques that can be applied in solving it. Most parallel search related problems fall into either loosely synchronous or asynchronous categories (31:924-926). This means that the size and number of tasks to be solved cannot typically be determined (with any accuracy) prior to execution. To some degree, extensions of standard data decomposition techniques can be applied to these problems, but Mudge and Abdalrahman's study of several different problems indicates that speed-up, and thus efficiency is often very low. Other techniques, such as the different variations of the controller-worker approach, have been shown to produce much better speed-ups than extensions of the static data decomposition approaches (1:1498). These approaches rely on beginning execution with an initial decomposition of the data set and then repartitioning the data set as execution proceeds based on the current workload of each processor. This repartitioning activity is known as *dynamic load balancing*.

The N-queens problem involves the placement of N chess queens on an $N \times N$ chess board so that none of the queens are in a position to directly attack another queen; Figure 3.5 shows one possible solution to a 5-queens problem. Finding all possible legal placements of the queens involves an exhaustive state-space search which can be implemented as a parallel search process. One possible way of performing the search in a well defined manner is start with an empty board and place one queen in an arbitrary row of the first column, such as row 2 of column 1 in the example. A separate *sub-problem* for each possible placement of a queen in the first column is therefore generated. The placement of the first queen in each sub-problem then defines the rows of the second column where the next queen can be placed. At this point, each sub-problem is further divided into more sub-problems which consider not only the placement of queens in the first column, but also the queens in the second column. For each successive "level" of sub-problems, the process

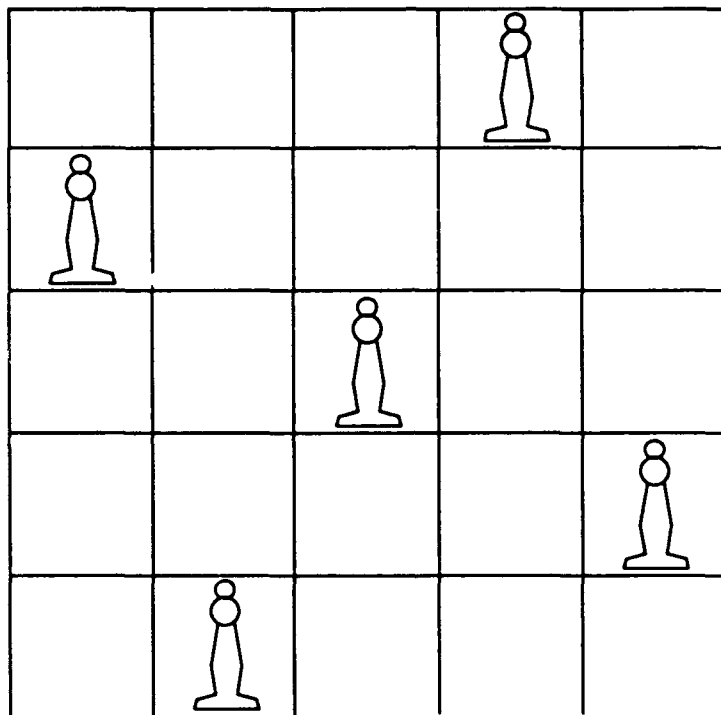


Figure 3.5. Example solution for the 5-queens Problem

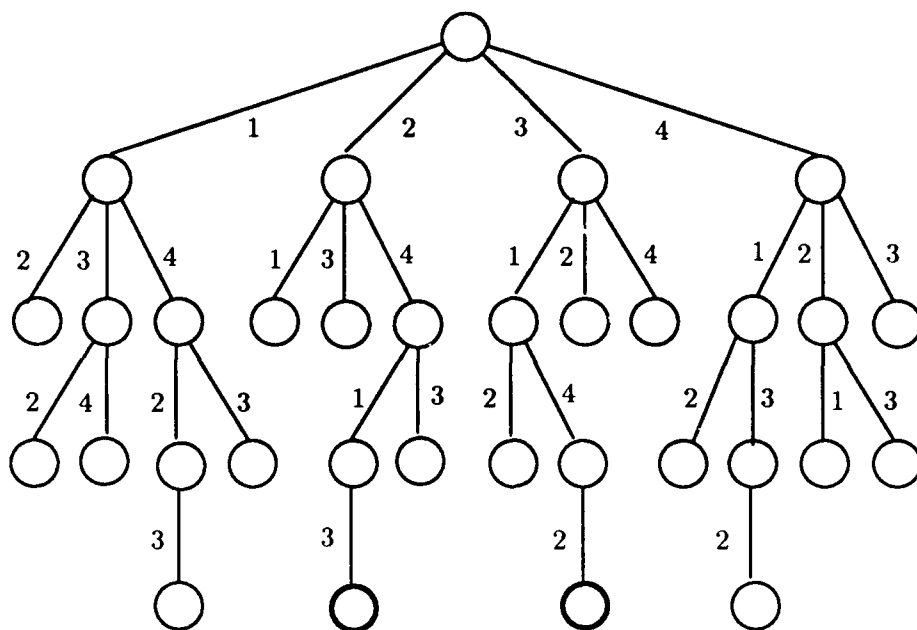


Figure 3.6. Search Tree for the 4-Queens Problem

of queen placement is continued until a column where a queen cannot be legally placed is encountered (no solution) or N queens have been successfully placed (a valid solution). The search tree in Figure 3.6 illustrates how the search tree is generated for a 4-queens problem as the search progresses.

One straightforward way to solve the N -queens problem uses a breadth-first-search (BFS) method viewed from a parallel perspective. Figure 3.7 presents a variation of Chandy and Mitra's BFS UNITY design (15:450) modified for the solving the N -queens problem. This program is considerably more complex than the matrix multiply program, but is fairly easy to understand once the semantics of this particular representation are explained. The BFS control structure specifies that the state-space search is carried out one "level" at a time until a solution (or in this case, all solutions) are found. The variable k determines the current level of the search and that the search process examines all possible extensions of the current path in parallel (all at once) before the next level is searched. Each element of the path array represents a column of the "board" and holds the current state of a possible solution path based on what row the queen is placed in; the path vector for the solution in

Program BFS-N-Queen

```

declare  integer k, /* search level */
         integer path[0..N-1] /* possible solution vector */
         path  solutions[0..N!] /* array of possible path solutions */
initially /* begin a level 0 with no current solutions */
         k = 0 || <|| i : 0 ≤ i ≤ N :: solution[1].path[i] = 0 >
assign   /* for each level in the BFS search, examine all possible extensions */
         <|| i : 0 < i ≤ Nk ::
           <|| j : 0 < j ≤ N ::
             /* extend current path by 1 (value of i) */
             solution[Nk + i].path[k] := j ||
             /* copy over previously defined part of path */
             <|| n : 0 ≤ n ≤ i :: solution[Nk + i].path[n] :=
               solution[Nk-1 + i].path[n] >
           if /* the following constraints are met: */
             /* A path to this point is fully defined and */
             <^ : (solution[Nk + i].path[k-1] ≠ 0 ∨ k = 0)
             /* new placement does cause a possible attack */
             ¬ attacking(solution[Nk + i],k,i)
           >
         > ||
         k := k + 1 if k < N
end {BFS-N-Queen}

```

Figure 3.7. N-Queens BFS UNITY Program

Figure 3.5 is {2, 5, 3, 1, 4}. The array of possible paths, solutions, is updated only when the current extension to the path is legal according to the guidelines of the N-queens problem and there is a valid path to that particular point in the search path. These conditions are contained in the “if” part of the large synchronous assignment statement.

This example is more complex than the matrix multiplication not only in terms of the UNITY representation, but also in terms of the problem decomposition. Given the search graph for the 4-queens problem in Figure 3.6, one possible decomposition approach involves dividing the search graph into separate sub-trees, each of which can be searched

independently⁶ At first blush, this might appear to be an easy solution to the problem, but past research shows that it is not always the most efficient one (69:89). Note that in the example problem, the major sub-trees have different numbers of search nodes to expand; two sub-trees have 7 nodes to expand, while the other two sub-trees have 9 nodes to expand. Therefore, using static decomposition techniques for this particular problem will certainly result in some degree of overhead due to load imbalance. In the case of the 4-queens example, the search tree is fairly symmetrical and the partitions are not badly out of balance. Many other search problems however, result in a highly irregular search tree that will result in load imbalance situations that are much more severe (1:1492). Often the static partition of the search tree containing the best solution will require the expansion of many more search nodes than partitions on other processors that do not contain the solution. The imbalance in the number of search nodes expanded means that some processors will complete their tasks long before others and will be unable to "help" the processors still working due to the static nature of the problem decomposition. Thus, search problems that result in more irregular search trees require a vastly different approach in order to produce efficient results.

Another familiar data decomposition method uses variations of the *controller-worker* approach⁷, which can include dynamic load balancing, to solve irregular problems more efficiently. The controller-worker approach uses one or more processors to "manage" the run-time partitioning of the data into fragments and to assign the computation associated with these various fragments to different processors. The three basic variations of the controller-worker approach are (1:1496):

- *Distributed List Approach* - In this approach the control process merely takes the original problem and breaks it into roughly equal size pieces and assigns a piece to each worker processor. The control processor then waits for all processors to complete

⁶This decomposition requires that the || symbol in the first part of the assign section (indicating synchronous operation) be replaced by a [] symbol (indicating asynchronous operation) which is valid for this particular problem.

⁷There are arguments as to whether this approach constitutes data or control decomposition. Although all processors no longer run the same problem, the emphasis is still on breaking down the actual data set, therefore the controller-worker approach will be referred to as a data decomposition technique

their part of the task. This approach is basically the same as the static decomposition approach discussed in the previous paragraph.

- *Central List Approach* - In this approach, the control processor holds a central queue of tasks to perform (in the case of the N-queens example, expanding a search node). A worker processor: obtains a task from the controller, performs the task and then sends any new tasks resulting from the execution of the assigned task back to the control processor. The control processor: enqueues new tasks sent any worker processor and continues the assignment of tasks to worker processor that have completed their currently assigned tasks.
- *Distributed List with Load Balancing Approach* - This approach is very similar to the distributed list approach except that any processor that is completing its initially assigned task can request new tasks (search nodes to expand) from another worker processor. When a worker processor with tasks remaining receives a work request from another worker processor, it sends part of the remaining tasks on its local queue (list) to the requesting processor. In this way, the remaining tasks are dynamically shared between all worker processors.

Dynamic load balancing techniques do have some associated costs requiring careful treatment in order to avoid unnecessary overhead during program execution. Both central list and load balancing approaches require communication overhead not associated with the static decomposition of the distributed list approach. In the central list approach, worker processors may be forced to wait for access to the queue because other processors are either enqueueing (posting) or dequeueing (obtaining) tasks. Delays resulting from contention for a shared object such as the central queue is a condition known as *bottlenecking*. In the load balancing approach, additional communication overhead occurs whenever work is shared between two processors. A minimum of two communications is required each time work is shared because idle processors must request work from another processors and the processor receiving the request must respond with either a "no work available" message or more data to be processed.

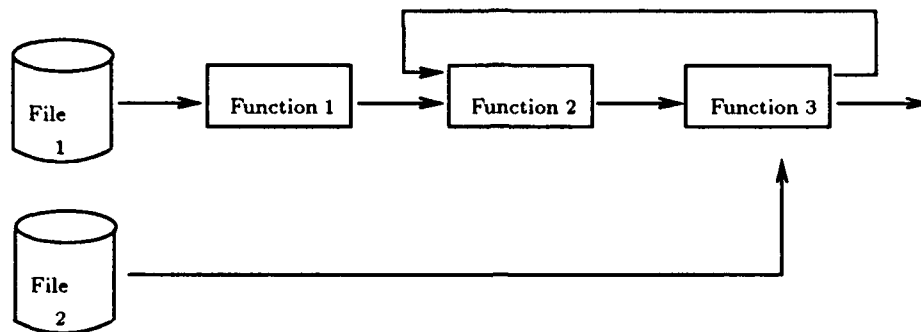


Figure 3.8. A Generic Static Control Decomposition Example

3.3.2 Control Decomposition and Generic Examples In the control decomposition approach, a problem is decomposed into a number of separate functional units instead of breaking up the problem based on the data that describes the problem. The control structure of a program, the algorithm, can often be viewed as a series of functional modules as shown in Figure 3.8 (4:6-1). In a sequential application, each of the modules is usually executed in sequence once for every distinct piece of data to be processed. In a parallel implementation, each processor performs a separate function on one or more stream(s) of input data sent to it by a data source (processor, file, keyboard, etc.) and outputs one or more streams to a data sink (processor, file, screen, printer, etc.). This process of communicating data between a series of independent processes is physically similar to instruction pipelining done in even simple modern microprocessors and pipelining of vector operations done in many supercomputers such as the Cray-1 (30:25). Operations in instruction and vector pipelines are typically only a very few computer clock cycles and therefore constitute very small grain synchronous tasks. On the other hand, control decomposition typically concentrates on problems that can be decomposed into a number of larger grain functions that can be performed asynchronously (4:6-3).

Based on the specific problem to be solved, it is often very difficult to divide a complex algorithm into a number of functions that require approximately the same amount of time. Additionally, certain "chunks" of data passing through the pipeline of functions may not all be the same size or require different amounts of processing time "in" each function. For instance, in Figure 3.8, function 1 must wait for function 2 to complete processing on

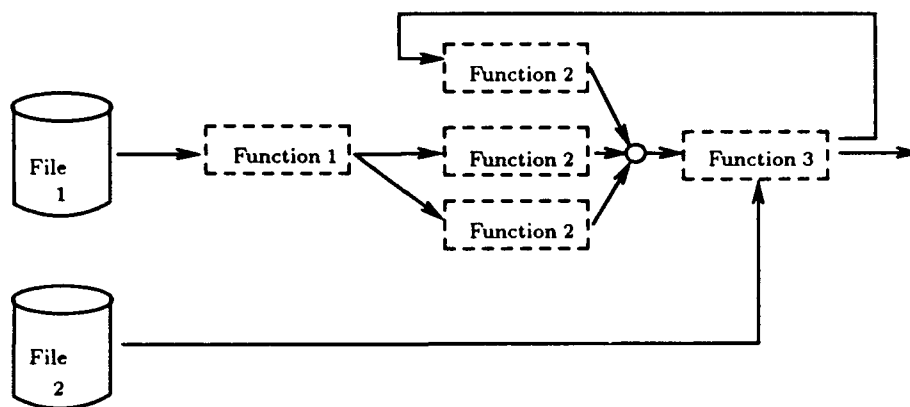


Figure 3.9. A Generic Dynamic Control Decomposition Example

its current "chunk" of data before it can begin processing on more of input file 1 or it can simply put the data in a queue that function 2 can access later. Whichever way the problem is approached, overhead stemming from task synchronization is likely to result. A dynamic load balancing approach to this generic control decomposition could significantly increase parallel program performance.

Figure 3.9 shows a generic example of how dynamic load balancing can be applied to a control decomposition type of problem. In this model, logical processes are created and assigned to physical processors as the need for a given function is detected. For example, when the processor executing function 1 finishes processing input file data, it begins running another process, say function 2. The use of task queues between processes allows execution of the parallel program as a whole, to continue without as much synchronization overhead. The downside of the use of dynamic processes is the acknowledged overhead cost associated with creating, terminating and scheduling logical processes and of course the cost of communicating (commonly referred to as piping) data between processes (3:18) (65:385). Problems using dynamic logical processes must possess large grain functional blocks and the operating system must be able to manipulate logical processes efficiently, otherwise control and communication overheads are likely to significantly reduce available speed-up.

3.4 *Parallel Computer Architectures*

After presenting a framework for representing problems, we must now concentrate on the detailed structure of the machines that will be used to solve them. This part of the chapter presents a brief overview of parallel computer architectures and classifies them with regard to their problem solving capabilities. The introduction of this chapter presents the essential characteristics of parallel computer architectures, but again these characteristics are: multiple processing elements and a means for allowing communication between processing elements. Beyond the two characteristics cited exists a tremendous diversity of computer architectures for parallel processing. It is not the intention of this thesis to present a description of every available parallel architecture, but instead to concentrate on general characteristics of commercially available, general purpose parallel architectures. The concepts of granularity, connectivity and homogeneity applied to classifying problems in the previous section also play a significant role in classifying parallel computer architectures.

3.4.1 Choosing a Useful Subset of Parallel Architectures In 1966, Flynn proposed a taxonomy based on the concept of instruction streams and data streams that is still widely used even more than two decades later. The four basic classes of computer according to Flynn's taxonomy are (26:1901):

- *Single-Instruction-Single-Data-Stream (SISD)* - a computer with only one central processing unit that operates on only one stream of data at a time, it is the architecture of a typical sequential computer.
- *Multiple-Instruction-Single-Data-Stream (MISD)* - a computer capable of performing multiple operations on the same data stream at the same time, an architecture that is generally considered infeasible.
- *Single-Instruction-Multiple-Data-Stream (SIMD)* - a computer that can perform the same operation on a number of different data streams at the same time (in a synchronous manner).

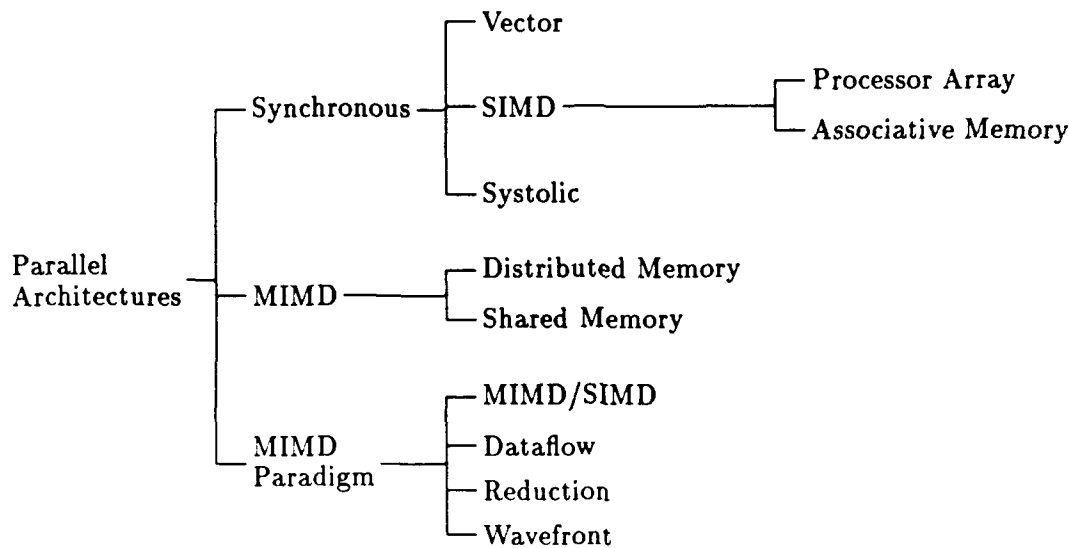


Figure 3.10. High-Level Taxonomy of Parallel Architectures (24:6)

- *Multiple-Instruction-Multiple-Data-Stream (MIMD)* - a computer with multiple processing elements, each capable of performing different operations on different sets of data at the same time (asynchronously).

This taxonomy provides a useful method for classifying parallel architectures, but it is incapable of effectively classifying the recent diversity of parallel architectures (24:6). As a result, an extended taxonomy (Figure 3.10) of parallel architectures is presented which extends Flynn's taxonomy, but effectively and efficiently covers the new diversity of architectures. Many alternative taxonomies for classifying all parallel architectures exist, but this particular one was chosen for its overall elegance and simplicity. The extended taxonomy incorporates some very recent (and sometimes exotic) parallel architectures which may not be familiar to the reader and are not specifically relevant to this thesis. Before discussing any aspects of the extended taxonomy further, it needs to be reduced to the minimum useful size of the purpose of this thesis.

For the purpose of this thesis, discussion is restricted to a subset of the extended taxonomy based on the following criteria:

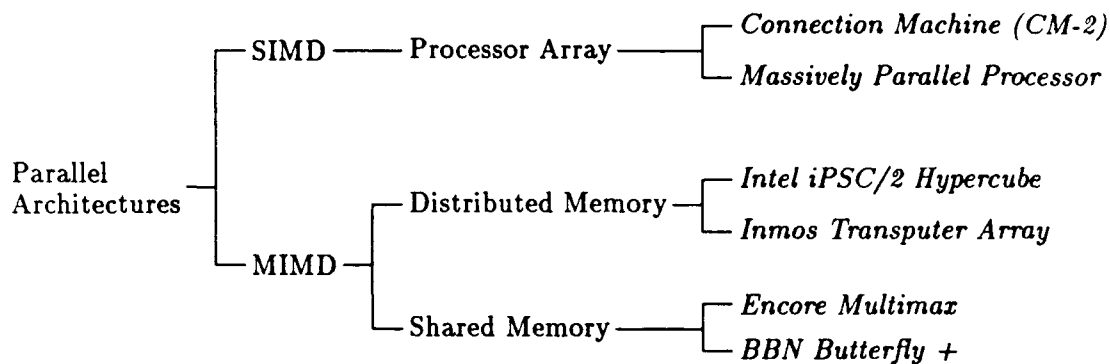


Figure 3.11. A Useful Subset of Parallel Architectures

- *General Purpose* - Architecture which are capable of solving a wide variety of problems, this criterion excludes machines which are specifically designed to solve just one class of problems.
- *Generally Available* - Architectures which are commercially available or experimental architectures which are widely available to researchers.
- *Known Performance* - Architectures for which important measures of performance are available. Among these important measures are: processor speed and time required for interprocessor communications.

Restricting the overall taxonomy to architectures meeting the criteria yields a selected subset of parallel architectures as shown in Figure 3.11. The elements in *italics* are specific examples of each major class/subclass in the taxonomy. Although this taxonomy excludes many new and promising parallel architectures, not enough information about the critical performance factors and problem solving capabilities of these architectures is currently known. Additionally, the UNITY approach for designing parallel programs does not directly support mappings to any of these architectures (15:82). A recent article by Ralph Duncan (24) provides a concise overview of the parallel architectures not specifically addressed in this thesis for readers interested in obtaining more information.

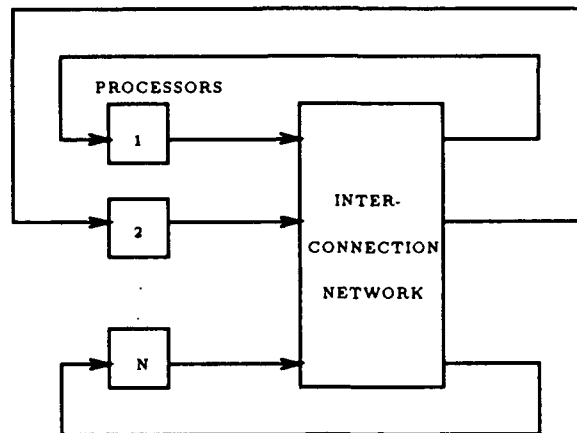


Figure 3.12. Architecture of a Generic Multi-computer (73:280)

3.4.2 MIMD Architectures In many ways, MIMD architectures are probably the most natural extension of the sequential computer architecture; therefore, this is where the discussion of specific parallel architectures begins. MIMD architectures are collections of independent computing elements that are capable of communicating with each other in some manner. Each MIMD computational element is capable of executing different sets of instructions (programs) on different sets of data. These processing elements are typically very powerful microprocessors that have access to a significant amounts of program and data memory (30:23). Collections of MIMD processors can be combined to solve problems through the use of interprocessor communications. The overall structure of the communications network is the main differentiator of the two classes of MIMD architectures. These two different structures are: distributed memory computers (also known as multi-computers) and shared memory computers (otherwise referred to as multi-processors). Hybrid architectures that incorporate both distributed and shared memory approaches also exist.

3.4.2.1 Distributed Memory MIMD Architectures Figure 3.12 shows the basic design of a distributed memory MIMD (DMM) architecture. Each independent processor, also called a node, contains a certain amount of local memory that only it can access. Communication between nodes takes the form of message passing, conceptually similar to

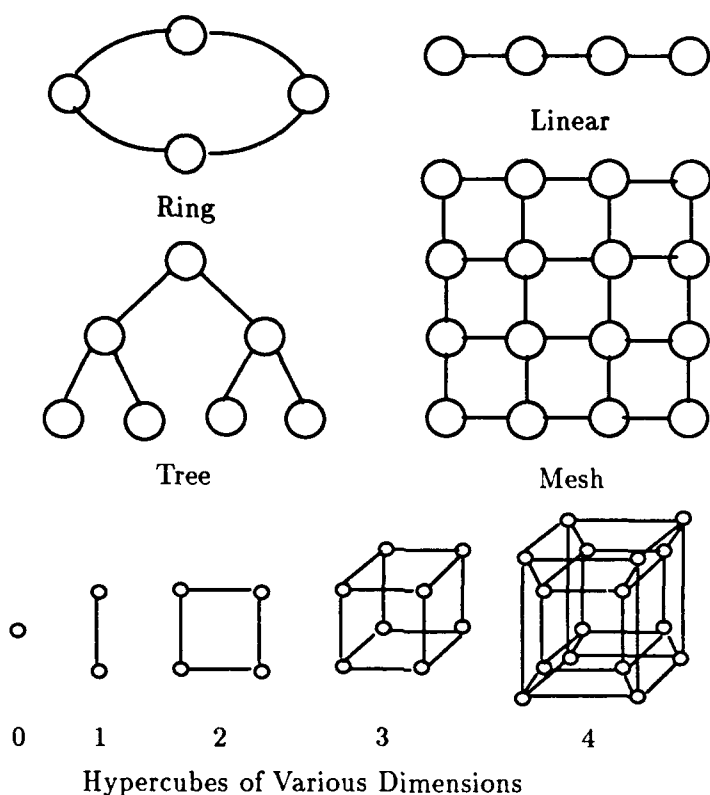


Figure 3.13. Distributed Memory MIMD Topologies (30:23) (24:10)

passing data packets between computers over a local area network. The node interconnection network can take many forms and the selected form determines the *topology* of the particular DMM implementation. Figure 3.13 shows several of the more popular choices of topology. The current philosophy of multi-computer design is to effectively hide the details of the topology from the programmer by giving the illusion that every node is directly connected to every other node in the computer (53:651). In reality, complete connection networks are not feasible for large numbers of processing elements (24:12), therefore multi-computer designers concentrate on efficient and scalable architectures. One of the major disadvantages of using DMM architectures is the significant communication overhead time (typically 10s of milliseconds) required to pass data between nodes. The size of communication overhead would appear to make DMM architectures incompatible with small grain problems; however, research to alleviate this problem is already well under way (21:2).

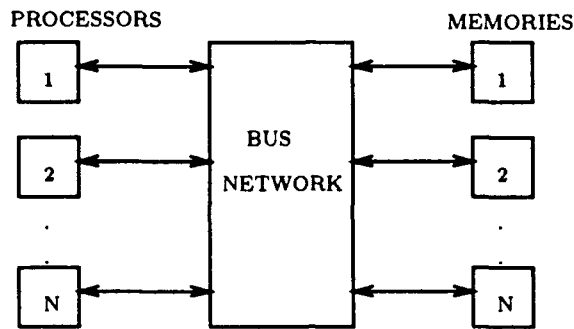


Figure 3.14. Architecture of a Generic Multi-Processor (73:280)

The use of efficient topologies and message passing techniques are vital in reducing the disproportionately large communication overheads typically suffered by DMM architectures. The binary N-cube (Hypercube) architecture shown in Figure 3.13 is one of the most popular multicomputer topologies, but some problems in the form of large communication overheads, inefficient message routing and the number of routing wires are evident when implementing machines with a large number of nodes (21:4). A newer network design based on a torroidal (end-connected) mesh attempts to adequately address these significant problem areas. MIT's torroidal routing chip (TRC) design uses shared memory, bidirectional parallel data paths and adaptive routing techniques to route a 150 bit message between nodes on a 1024 node processor in an average time of 7.5 microseconds (21:6). The TRC represents a several order of magnitude decrease in communication time over the 10s of milliseconds required for current generation multi-computers. Both Intel and Symmult (Amtek) have announced the intention to use the TRC in prototype designs for the next generation of multi-computer products. Therefore, it is likely that the next (third) generation of multi-computer will be much more compatible with smaller grain tasks.

3.4.2.2 Shared Memory MIMD Architectures Figure 3.14 shows the basic layout of the shared memory MIMD (SMM) architecture. In many ways, SMMs are very

similar to DMMs except as their name implies, communication between processors takes place through the use of shared memory instead of through message passing. In the SMM approach, each processor typically possesses some local memory that only it can use plus access to a significant amount of global (or shared) memory through some type of interconnection network. Figure 3.15 presents some of popular shared memory interconnection topologies that are being used in commercially available multi-processor implementations. These networks must typically have large bandwidths in order to minimize overhead resulting from CPUs accessing necessary instructions and/or data. As a result, data passing between processors using shared memory techniques is very fast (on the order of several memory access cycles). However, the greater the number of processors, the greater the contention for access to shared memory becomes. Shared memory contention is typically the greatest when using a bus interconnection network (73:302). The problems with SMAs is, therefore, not so much communication time, but the efficient scalability of the interconnection topology used.

Each topology in Figure 3.15 has both advantages and disadvantages with respect to variable access time and scalability. The bus interconnection network allows very fast access to shared variables, but 20 processors is the acknowledged limit for a single shared bus system (73:283). Multiple busses connected by cache memories are an alternative to single bus designs, but this concept is still in the experimentation stages (3:421). Crossbar switches allow for complete connection of processors and memories which provides effective performance, but this approach is generally very costly and not considered feasible for highly parallel architectures because the number of switching elements required is $O(N^2)$, where N denotes the number of processors in the parallel architecture (24:12). Multi-stage switches, like the Omega network, requiring $O(N \log_2 N)$ switches, provide scalability, but suffer from relative increases in memory access time as the number of stages increases. Using a six stage multi-stage network, BBN's Butterfly+ requires only slightly more time to access a remote memory location (≈ 6 microseconds), than to access a local memory location (≈ 2 microseconds) (3:456). Considering the scalability of the multi-stage networks and their relatively fast access times, this increase in access time to add considerably more processors might appear cheap. It is very difficult to make value judgments about the

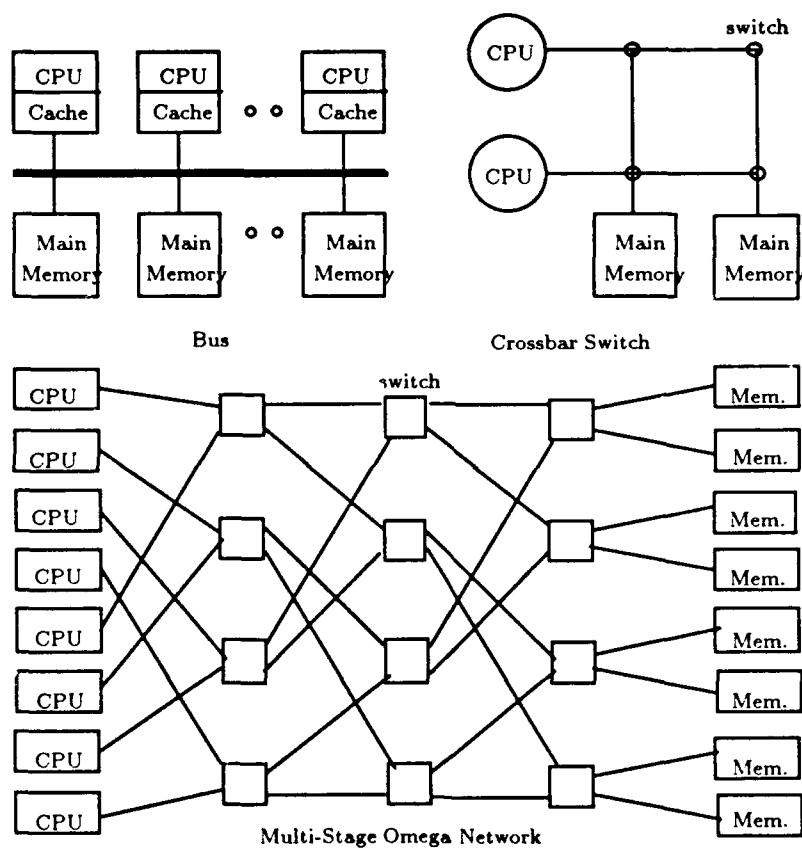


Figure 3.15. Architecture of a Generic Multi-Processor (73:158)

“best” approach to connecting multi-processor machines without more information on the intended class of applications. The “best” SMM topology depends on the required scale of parallelism and the acceptable limits on shared memory access times.

3.4.3 SIMD Processor Array Architectures In many ways, the concepts embodied by SIMD architectures make them as natural an extension to sequential computers as MIMD architectures; it is possible to abstract SIMD architectures as sequential architectures that are simply operating on a much larger number of bits. SIMD architectures are usually implemented on a scale more massive than other parallel architectures using multitudes of efficiently connected, simple (but often fast) processing elements (30:22). Figure 3.16 illustrates the basic architectural components of a SIMD array architecture and the connections between those components. In addition to a simple ALU and control circuitry, each processing element also typically possesses a small amount of local memory (less than 1000 bytes), and facilities for communicating with neighboring processing elements. The limited memory on each processing element is not sufficient to contain a computer program of any complexity, therefore instructions must be provided by the control computer. In the SIMD mode of operation, each processor executes the instruction broadcast to it by the central control computer. All processing elements are connected to the same system clock, therefore the operation of all processors is completely synchronous. The large number of processing elements provide the potential for parallelism on a massive scale, but only if the problem being solved “cooperates” by being homogeneous on an equally massive scale.

MIMD based architectures often have a very difficult time achieving speed-ups when executing small grain tasks due to communication overheads, but SIMD machines are capable of producing significant speed-ups with small grain tasks because of their efficient processor to processor communication structures. One or more of the interconnection topologies used for multi-computers (Figure 3.13) can also be used to connect SIMD processing elements. On many SIMD machines, communicating data between neighboring processing elements often requires only one instruction cycle (3:339). Some SIMD implementations restrict data exchanges to nearest neighbor elements, but others (such as the CM-2) allow for communication between processors that are separated by some power of 2

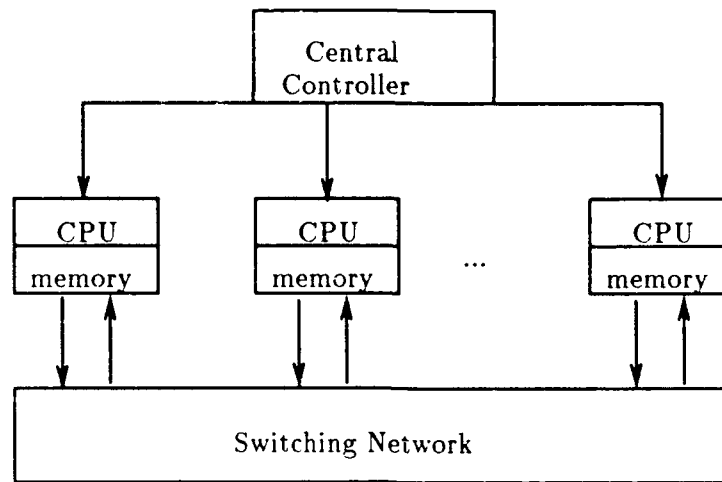


Figure 3.16. General Structure of a SIMD Array Architecture

other processing elements (3:344). This "remote" communication typically requires more than one machine cycle, but is more efficient than passing the message through 2^N nearest neighbor processors (3:348). Efficient passing of data between processing elements means that very few operations on each piece of data is required to "amortize" communication cost. Consequently, significant speed-ups for small grain size messages can be achieved much more easily than with most MIMD machines if the problem contains sufficient homogeneity.

Most SIMD architectures have the inherent capability to solve very small grain problems efficiently (30:23), but pay a price for this ability in terms of capability to solve inhomogeneous problems. Because only one instruction can be executed at a time, unused processors "left over" from performing computations on data sets smaller than the number of processors cannot typically be used to perform other tasks. Efficiency, and therefore speed-up, decreases as the ratio of processors used to total processors decreases. The size of the homogeneous data set is directly proportional to the number of processors used and both decrease as the problem becomes more and more inhomogeneous. Therefore, prospects for efficient use of SIMD architectures will generally fall off rapidly as the homogeneity of the problem decreases.

A particular problem does not require complete homogeneity because SIMD architectures can use certain techniques to “generalize” the affect of global instructions. The data contained on each processing element is capable of affecting how a given instruction is actually executed: Each processor node typically has its own set of status registers, similar to flag registers on most modern microprocessors. Local processor flags can be used to modify that processing node’s response to subsequent instructions (30:22); for example a *skip-on-zero* instruction causes any processing element with a set zero flag to skip execution of the next instruction. Another approach to generalizing the affect of global instructions is through the use of *very long instruction words* (VLIWs). Each VLIW instruction actually contains a number of instructions that can be used to control different partitions of SIMD processing elements (3:478). Under the VLIW concept, each instruction is still executed synchronously, but subsets of SIMD processing elements can now execute different instructions. Use of data dependency control and VLIWs may allow SPA architectures to be more flexible and thus more able to handle less than totally homogeneous problems, but in all likelihood, they will never have the capability to efficiently solve truly asynchronous problems.

3.5 Summary: Problems and Architectures, A Common Framework

Given a specific problem and a specific means for solving it, there will almost always be one parallel architecture that produces the best performance in terms of speed-up; however, this same architecture may not produce the best performance in terms of efficiency or even execution time. As with many other applied science fields, tradeoff decisions are almost always necessary when “mating” a problem to a specific architecture. Figure 3.17 presents a very general common framework for problems and architectures based on extensive research into the characteristics of parallel problem solving by Geoffrey Fox (31). The positioning of the architectures on the problem space background indicates that a given architecture has been proven to provide efficient solutions for problems of a particular type. Although the UNITY approach supports development of correct programs, a priori information about the relation of problem characteristics to parallel architectures is required. Without this information, the parallel program designer is unable to make

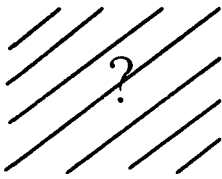
Problem	small grain	medium grain	large grain
Synchronous	MIMD & SIMD		MIMD & SIMD
Loosely Synchronous	SMM 	MIMD	MIMD
Asynchronous		SMM	MIMD

Figure 3.17. A General Common Framework for Problems and Architectures (31:953)

intelligent decisions about how to map the problem to a particular architecture. It also provides some feel for whether implementing a particular problem on a specific architecture is likely to provide efficient results.

IV. Quantifying Parallelism in Production Systems

4.1 Introduction

At first appearances, production systems appear to possess large amounts of parallelism, but as noted in chapter 1, this parallelism is severely constrained by the characteristics of a given application (36:1). This chapter draws upon the concepts of the two previous chapters to present ways in which parallel processing can be used to increase the execution speed of production systems. It also explores the limits on increasing the execution speed imposed by the characteristics of specific applications. Like the example problems presented in Chapter 3, the amount of parallelism and the nature of that parallelism changes dramatically depending on the problem decomposition. In this chapter, I introduce a hierarchy of possible production system decomposition techniques. Most of the decomposition techniques discussed are basically implementation-dependent, but some are dependent on specific aspects of the Rete algorithm for the reasons discussed in Chapter 2, Section 7. For each level in the hierarchy, the amount of useful parallelism is examined, not just in qualitative, but also in quantitative terms. The metrics used to quantify the amount of parallelism for a given decomposition provide the basis for development of the STEPPS system described in Chapter 5.

As in the simple matrix multiplication example in Chapter 3, Section 3.1.1, this chapter takes a *top-down* approach to investigating and quantifying available parallelism. The *top-down* decomposition begins with the original sequential task and then iteratively divides it into a greater number of finer grain tasks. The *top-down* approach yields the hierarchy of decomposition techniques alluded to in the previous paragraph. A problem decomposition that is derived directly from the original problem constitutes *high-level parallelism*, whereas a decomposition that relies on a previous decomposition is termed *low-level parallelism*. Figure 4.1 provides a visual representation of the *levels* of parallelism that might exist in a generic type problem. It shows how the problem of computing a number of discrete elements can be decomposed into relatively large grain tasks, each of which can be broken down several smaller size tasks. For the most part, the emphasis is on data decomposition instead of control decomposition, because data decomposition provides

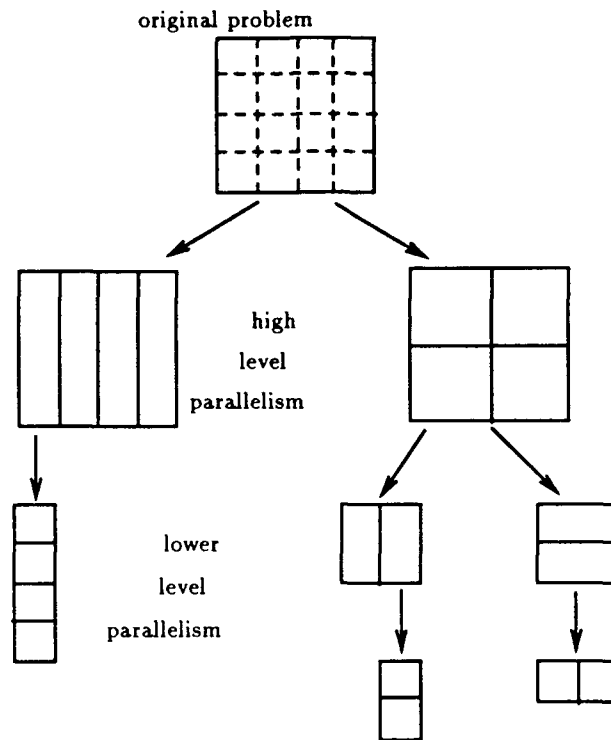


Figure 4.1. A Hierarchy of Problem Decompositions

the most promise for achieving high levels of parallelism (31:903). Using the "levels of parallelism" concept, investigation of the amount of parallelism in a given production system application can be approached in a more orderly manner.

4.2 Sources of Parallelism

Production systems provide numerous opportunities for parallelism through decomposition of data structures (rule base and working memory) and the control structure (match-select-act cycle), but they also possess a number of attributes that conspire to limit available parallelism. As illustrated by Figure 2.1, production system execution takes place in cycles where the next cycle cannot begin until the tasks of the current cycle are completed. In every production cycle, there exists a synchronization point separating subsequent production cycles between the conflict resolution and the act phases (36:43). The synchronization point occurs here, because the chosen rule to be executed must be known before the next act phase can commence. This mandatory synchronization point

means that parallel processing must take place within a given production cycle. Thus, for a majority of this chapter, the levels of parallelism within a production system task are discussed in the context of the sequential task of executing a single production cycle.

Within a production cycle, there are three sequential tasks (match, select and act), but as shown in Chapter 2, Section 5, the match phase is by far the most computationally expensive of the three production cycle phases. Although parallelism exists in both select and act phases (36:53), the greatest increase in execution speed can be gained through applying parallelism to the match phase (70:1346). Theoretically, it may be possible to overlap execution of the three phases of a production cycle in a pipeline type fashion (see Chapter 4, Section 3.4), however the match phase constitutes a serious bottleneck in the execution pipeline (36:51). For this reason, this investigation does not directly address pipelining of data through the processes of the production cycle, but instead concentrates primarily on data decomposition. Unless specifically stated, the decomposition will be relative to the task of executing one production cycle.

The UNITY representation of a simple production system shell in Figure 4.2 provides the basis for extracting and quantifying most of the levels of parallelism. The shell represented by this meta-level program has several restrictions that normal production system shells do not. Some of the more significant simplifications are: it allows only predicates with constants to be matched, asserted and retracted and matching is done using exhaustive search instead of using the Rete algorithm. Pattern matching using variables is one of the most powerful facilities of production systems, however the inclusion of variable binding would have added unnecessary complexity to shell's design (18) (17). Similarly, the reason for not using the Rete match algorithm is due to its underlying complexity and difficulty in accurately representing it using the UNITY formal language. This simple shell program (design) only provides a basis for displaying and discussing the various alternatives associated with parallelizing production systems, it is not the sole basis for evaluating parallelism; this is particularly true for decomposition methods in which the Rete algorithm is involved. Instead of rewriting much of the program in Figure 4.2, the decomposition approaches will merely modify the contents of certain program lines.

Program Simple-Shell

```

1      declare integer entry /* number indicating current conflict set entry */
3      predicate char[1..max], /* string representation of a fact */
4      function {assert,retract}, /* possible functions on WM */
5      rule      record /* record of elements making up a rule */
6      lhs_size natural, /* number of predicates in the lhs */
7      lhs predicate[1..lhs_size], /* lhs predicates */
8      rhs_size natural, /* number of predicates in the rhs */
9      rhs predicate[1..rhs_size], /* rhs predicates */
10     rhs_func function[1..rhs_size], /* actions on rhs predicates */
11     end record;
12     Rule_Base      rule[1..R]; /* Array of rule elements */
13     Fact_List      rule[1..F]; /* Working Memory (WM) elements */
14     Agenda         rule[1..max]; /* Set of rules to be executed */
15 initially /* Assume rule base and fact list already loaded */
16 Agenda, entry = NULL, 0
17 assign /* Iteratively perform the Match-Select-Act cycle until done: */
18 /* Match phase: put most recently satisfied rule on Agenda */
19 <|| r : 0 < r ≤ R ::
20     Agenda := Rule[entry] || entry := entry + 1 if
21     /* A rule is satisfied if, for all predicates in the rule, */
22     <|| ∧ : 0 < a ≤ Rule_Base[r].lhs_size ::
23         /* there is at least one fact in WM that matches */
24         <|| ∨ : 0 < f ≤ F ::
25             Rule_Base[r].predicate[a] = Fact[f] >>
26     > if Agenda = NULL
27     /* Select phase: the agenda already contains most recent rule */
28     /* Act phase: carry out all actions specified by the agenda's rhs: */
29     <|| c : 0 < c ≤ Agenda[entry].rhs_size ::
30         /* Delete any element of WM matching a fact to be retracted */
31         <|| f : 0 ≤ f < F :: Fact_List[f] := NULL if
32             Agenda[entry].rhs_func[c] = retract ∧
33             Agenda[entry].rhs_args[c] = Fact_List[f] >> ||
34         /* Put an asserted fact in the first NULL location in the fact list */
35         <|| f : 0 ≤ f < F :: Fact_List[f] asserted := NULL, FALSE || asserted if
36             Agenda[entry].rhs_func[c] = assert ∧
37             Agenda[entry].rhs_args[c] = NULL ∧
38             ¬asserted ∧
39         >> || Agenda, entry := NULL, 0
40     > if Agenda ≠ NULL
41 end {Simple-Shell}

```

Figure 4.2. UNITY Program for a Simple Production System Shell

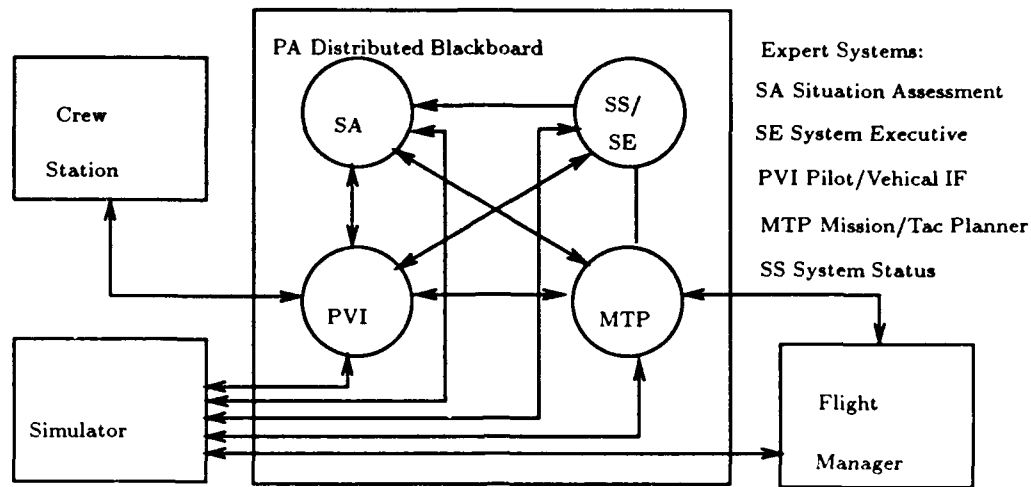


Figure 4.3. Pilots Associate, Top Level Architecture (78:4)

4.3 Application Parallelism

Application parallelism is exploited by systems that use several cooperating, but loosely synchronous production system based expert systems executing in parallel (36:54). This particular level of parallelism departs from the original assumptions of the type of parallelism being investigated; therefore, it is discussed first. Application parallelism represents a level of parallelism higher than a production cycle task, the earlier stated basis for this investigation into available parallelism. Additionally, this level of parallelism typically arises in the context of a complex system containing two or more relatively independent expert systems which is akin to functional, not data decomposition. The *cooperating experts* making up the system typically communicate with each other through a shared data space known as a *blackboard* (54:561). An example of using application parallelism is the Pilot's Associate, an expert system to assist the pilot of a single crewmember fighter or attack aircraft with navigation, threat-avoidance and other mission related tasks (78:1). Figure 4.3 shows that the Pilot's Associate system is actually composed of four expert systems communicating through a distributed (as opposed to a central) blackboard. The application parallelism approach may appear to be a very natural way of implementing a complex system, but there are some significant drawbacks associated with this approach.

4.3.1 Measuring Application Parallelism Reviews of previous research reveal that significant increases in execution speed are possible using application parallelism, but also that significantly increased parallelism is not always the rule (36:54). It is fairly obvious that the maximum amount of speed-up to be gained from using application parallelism is limited by the number of cooperating experts which, from available literature, appears to be very small for most applications (36:54) (78:2) (57:1327) (52:29-35). Very few production systems can effectively achieve the maximum attainable level of parallelism because they were originally designed to run on single processor computer architectures (36:54). Systems designed to run on distributed blackboard architectures, such as the Pilot's Associate face the problem of effectively partitioning the functional tasks among a number of processors. Application parallelism is treated as a static decomposition method which prevents a processor not currently performing any tasks from assisting a processor that may be fully loaded with tasks. Thus static load balancing is very likely to result in loss of parallelism due to load imbalance. Additionally, when one cooperating expert requires information from another, it may become idle until that information becomes available resulting in synchronization overhead and, subsequently, an additional loss in speed-up.

4.4 Agenda or Rule-Level Parallelism

According to the strict definition of the production system paradigm, only one rule is executed every production cycle, but Stolfo and Ishida note that it is possible to execute more than one rule in parallel without affecting the correctness of the application (48:586). The agenda is another name for the set of rules currently in the conflict set. Therefore, the execution of more than one rule during a given production cycle is referred to as *agenda parallelism*. Applying agenda (or rule-level) parallelism has received little attention and it is probably the least well understood aspect of parallelizing production systems (23:78). Abstractly, agenda parallelism attempts to "compress" the processing of a number of rule executions into a single production cycle instead of spreading the same processing over several successive production cycles. Executing multiple rules during a single cycle seeks to increase the number of tasks available between mandatory synchronization points (between the select and act phases) thus allowing potentially greater amounts of parallelism. Figure

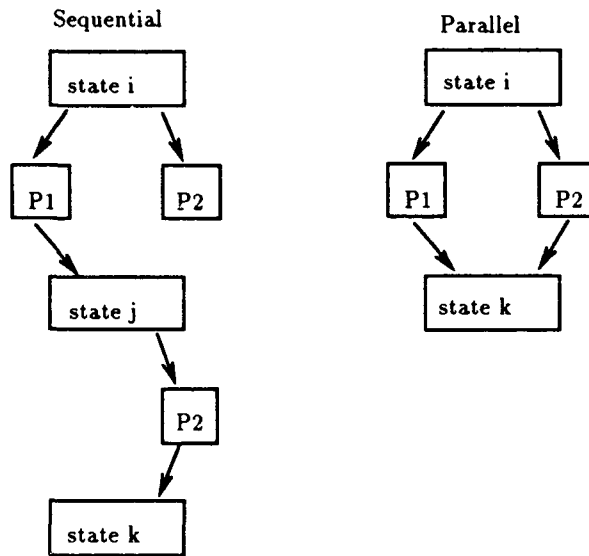


Figure 4.4. A Simple Comparison of Sequential and Parallel Rule Firing

4.4 provides a visual representation of the “compressing” affect of executing two rules (P1 and P2) in parallel. Viewing this figure as a Gantt chart reading down the page, it can be observed that agenda parallelism has the potential to decrease the execution time of certain production system applications if the applications themselves are amenable to agenda parallelism.

Most production systems applications are designed with sequential execution in mind. However, evidence suggests that agenda parallelism may still provide substantial benefits. Whenever an application contains separate threads of execution or must review a number of possible solution alternatives, the potential for the effective use of agenda parallelism exists. For instance, blackboard systems contain a number of communicating expert systems which pursue separate threads of execution in parallel. By combining these expert systems into a single, tightly coupled system, agenda parallelism can be used to maintain parallel execution of these separate threads of control. The workload associated with working memory updates can also be more easily divided among the available processors thus overcoming the significant load imbalance problem in statically partitioned blackboard systems. Potential agenda parallelism can also arise in the context of a “difficult-to-define” search problem. The Digital Equipment Company’s (DEC’s) XCON expert system, for

configuring VAX computers with customer specified options, typically evaluates a number of possible computer system configurations. Different system configuration possibilities need to be explored because "there is no suitable evaluation function that can be applied to say a partial configuration C is better than another partial configuration D. Sometimes the XCON system must generate more than one complete computer system configuration before a single best configuration can be chosen" (49:135). Searching for several alternative results also provides significant possibilities for achieving speed-up using agenda parallelism.

A review of past research reveals only two production system shell variants that are capable of exploiting agenda parallelism: Carnegie Mellon's Soar interpreter (76) and Merit Technology's Merit Enhanced Traverse Engine (METE) system (51); however, both of these system deviate from the traditional production system paradigm in some important ways. The METE system uses only rule and data sets that cannot "interfere" making it impractical for use with a majority of current production system applications. The METE system is covered in more detail in the section on Node parallelism later in this chapter. In contrast, the Soar formalism is used to emulate learning as well as intelligent problem solving through a process called *universal sub-goaling*. The Soar formalism is similar in many ways to the production system formalism, but allows parallel rule firing. In each phase of the Soar interpreter cycle (shown in Figure 4.5), all satisfied rules are fired. However, the learning mechanism in the Soar formalism tends to be computationally expensive and deviates significantly from the traditional production system formalism (76:2). Therefore, Soar will not be considered in the context of rule parallelism. Soar's concept of adding rules during run-time is also important to the discussion of production parallelism in the next section.

4.4.1 Characteristics of Agenda Parallelism Past research shows that satisfied production rules can be correctly executed during the same production cycle (in parallel) as long as they do not interfere with each other (77:57) (22:22). The following simple example illustrates the concept of non-interfering rules (62:149):

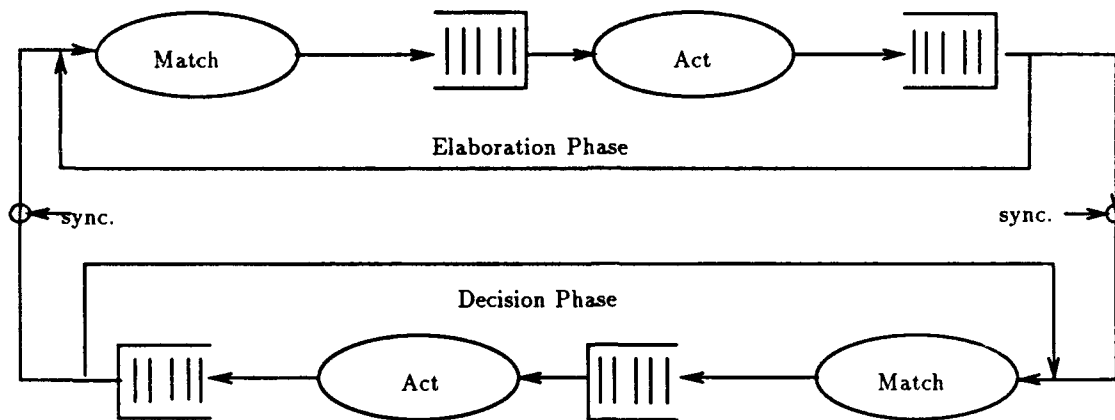


Figure 4.5. The Soar Formalism Interpreter Cycle (36:44)

Given the following two rules in the conflict set (agenda):

(RULE_A	(RULE_B
(MATCH_A)	(MATCH_B)
=>	=>
(ACT_A))	(ACT_B))

where the MATCH predicates are sets of rules in working memory
and the act predicates are rules to be added to or retracted
from working memory, the two rules are said to be independent
if and only if:

$$\begin{aligned}
 &((ACT_A \cap ACT_B = \emptyset) \wedge \\
 &(ACT_A \cap MATCH_B = \emptyset) \wedge \\
 &(MATCH_A \cap ACT_B = \emptyset)).
 \end{aligned}$$

The non-interference criteria above is conservative and may not detect all possible parallelism, but more testing is required to extract additional parallelism. When the conflict set contains more than two rules, each possible pair of rules in the conflict must be tested to ensure that non-interfering rules are not executed in parallel (62:149). The UNITY program fragment in Figure 4.6 shows how the original simple production system shell can

Program Agenda-Shell

```

1      declare
      :
15     initially /* Assume rule base and fact list already loaded correctly */
16           Agenda, compressed = NULL, FALSE
      :
17     assign  /* Iteratively perform the match-select-act procedures */
      :
27           /* Select phase: compress keeps only the largest possible set of */
           /* non-interfering set of rules on the agenda (see section 7.3) */
           < N := compress(Agenda) || compressed > if Agenda ≠ NULL and ¬compressed
           /* Act phase: carry out all actions specified by the agenda's rhs: */
           /* the actions of every rule can be executed in parallel (non-interfering) */
           < <|| r : 0 ≤ entry < N ::
29               <|| c : 0 < c ≤ Agenda[entry].rhs_size ::
           :
40               > || Agenda, entry, compressed := NULL, 0, FALSE
41           > if Agenda ≠ NULL ∧ compressed
end {Agenda-Shell}

```

Figure 4.6. UNITY Program for a Simple Production System Shell

be modified to achieve parallel rule firing. Chapter 7 expounds on the concepts of agenda parallelism correctness and methods for allowing maximum parallel rule firing.

4.4.2 Bounding Agenda Parallelism Using state saving algorithms (such as Rete), executing more than one rule at a time does not increase the execution speed of a production system implemented on a single processor computer. For example, consider two non-interfering rules P1 and P2 in Figure 4.4 each with 3 actions to perform. Additionally, assume that the cost of executing each action and updating the Rete network requires the same amount of time. Recall that the number of rules are non-interfering, therefore the order in which the rules are applied is truly non-deterministic. In a sequential implementation, three action tokens are “propagated” through the Rete network each production cycle for a total of 6 “token flows”. If both rules are executed in the same production

cycle, the cost is the same 6 “token flows” which is equal to two successive production cycle updates.

Although parallel rule firing is of no benefit to sequential implementations, parallel implementations are capable of effectively exploiting the effects of parallel rule firing. Given that the sets of actions to be executed are non-interfering, different processing elements can be used to execute each action on a single “global” Rete network simultaneously. The level of parallelism, and thus maximum speed-up, that agenda parallelism can provide depends on the number of rules that can be executed in parallel during each production cycle. Given a production system where \mathcal{R} rules of equal action cost can be executed in parallel each cycle, the upper bound on speed-up is:

$$S_{UB} = \frac{(M + S + A)\mathcal{R}}{M + A} = \mathcal{R} + \frac{S \cdot \mathcal{R}}{M + A}$$

where M is the match phase time, S is the select phase time and A is the act phase time. However, the number of non-conflicting rules that will be executed in parallel during any given production cycle cannot be determined prior to execution (48:572). Additionally, the processing time associated with a given rule execution cannot be determined except by actually executing the program (59:98). Although the level of agenda parallelism in a given application cannot be determined, even loosely except through execution, its potential impact on lower levels of parallelism is considered throughout the remainder of this chapter.

4.5 Production Parallelism

Production parallelism consists of dividing (or partitioning) an application rule base among a number of communicating processing elements which allows each processor to execute the match phase on a given subset of the rule base in parallel. Rule partitioning can be performed statically or dynamically, but dynamic methods are very likely to possess prohibitively high communications costs (59:96). Experiments with the Soar formalism show that adding productions during run-time and subsequently updating the network to reflect the current state of working memory is computationally expensive (76:7). Based on this research, using dynamic methods to increase speedup appears infeasible due to

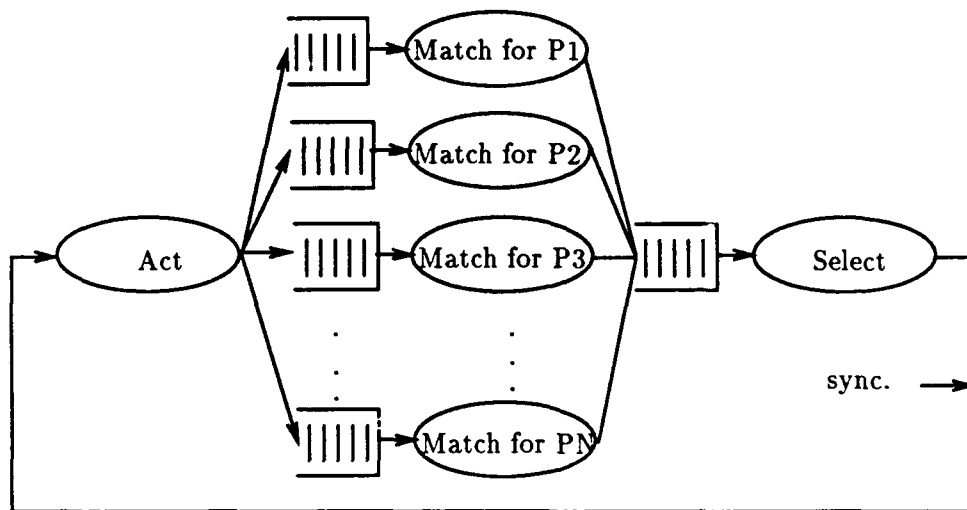


Figure 4.7. Production Parallelism (36:48)

the overhead required to redistribute rules and recompile the local Rete network on each processing element during each production cycle. Using static rule partitioning, rules in the rule base can be assigned to different processors prior to run-time thus eliminating the overhead associated with communicating rules and compiling the Rete network. Figure 4.7 illustrates how (static) production parallelism is used to divide a rule base into N partitions that are matched in parallel.

A review of past research in parallel production systems identifies a number of efforts aimed at implementing production parallelism on several different parallel architectures. A majority of the past implementations reside on multi-processor systems, but there are several efforts that concentrate on the use of commercially available and specially designed multi-computer architectures. Researchers at Carnegie-Mellon have produced less than 2 fold speed-up using a production parallelism implementation of OPS-5 on three processor VAX 11/784 and VAX 8200 multi-processor systems (60:2). Similarly, researchers at NASA's Johnson Space Center (JSC) have produced only 3.3 fold speed-ups using a parallel implementation of the CLIPS shell on a 4 processor FLEX/32 multi-processor (65:386). In contrast, multi-computer implementations have been less successful in producing speed-ups primarily due to heavy communication costs. The *HyperCLIPS* shell constructed for

the Intel iPSC/2 multi-computer was not able to produce any speed-up due to the prohibitively high overhead costs in comparison to relatively small processing requirements associated with the Rete match algorithm (42:8). The remainder of this subsection explores the basic concepts of production parallelism and why it does not appear to produce very significant speed-ups.

4.5.1 Characteristics of Production Parallelism According to Figure 4.7, only the match phase is performed in parallel whereas the select and act phases are performed sequentially. When using the Rete match algorithm, updating of the Rete network local to a given processor (analogous to the match process plus some of the act process) can be executed in parallel. However, in order to correctly perform the selection step of the production parallelism cycle, the list of satisfied rules from each processor must be brought together in a *global agenda* (65:385). From this global agenda, the single "most eligible" rule can be selected for execution thus preserving the original production system formalism. Additionally, the part of the act phase that binds variables, makes function calls and performs I/O should also take place on the processor containing the satisfied rule (65:385). The reasons for performing the selection and parts of the act on the same processor as the satisfied rule are:

1. it simplifies the refraction process and
2. because only the processor containing the selected rule contains the correct variable bindings.

The original production shell UNITY program assumes that all rule matches are carried out at the same time, but this approach does not allow less processing elements than the total number of rules to be used. Figure 4.8 shows how the original simple production system shell can be modified to include the production parallelism concepts when the total number of processors is less than the total number of rules in the system. In order to accurately represent production parallelism, the match phase is divided into a set of asynchronous match processes for each of P partitions of production rules. The UNITY program assumes that the number of rules can be evenly divided by the number

of partitions (processors) and that the rules have already been evenly distributed among these partitions (processors). The production parallelism modification also requires that each processor put the satisfied rules on a global agenda whenever a satisfied rule is found. The act phase is, for the most part, unaffected by the production parallelism modification as each processor maintains a copy of the contents of the WM.

Program Production-Shell

```

1      declare
      :
15     initially /* Assume rule base and fact list already loaded correctly */
      :
17     assign /* Iteratively perform the match-select-act procedures */
18           /* Match Phase Each partition performs match independently */
           <[ $p : 0 \leq p < P ::$ 
               <[ $r : \frac{R}{P} \cdot p \leq r < \frac{R}{P} \cdot (p + 1) ::$ 
20                 Agenda := Rule[entry] || entry := entry + 1 if
               :
27             /* Select phase: keep only the last rule put on the global agenda */
28             /* Act phase: carry out all actions specified by the agenda's rhs: */
           <[ $p : 0 \leq p < P ::$ 
               <[ $r : \frac{R}{P} \cdot p \leq r < \frac{R}{P} \cdot (p + 1) ::$ 
29                 <[ $c : 0 < c \leq \text{Agenda}[\text{entry}].\text{rhs\_size} ::$ 
               :
41             ] if Agenda  $\neq$  NULL
end {Production-Shell}

```

Figure 4.8. UNITY Program for a Simple Production System Shell

4.5.2 Bounds on Production Parallelism The static rule partitioning methods for applying production parallelism appear particularly inviting because no communication between processors is required during the match (or Rete update) phase, the phase typically requiring the most processing time (36:45). Given the apparently low communication requirements and the large computation requirements that can be processed in parallel, production parallelism appears to be a very promising method for increasing speed-up.

In reality, there are a number of factors that combine to limit available speed-up using production parallelism.

Conceptually, the maximum match phase parallelism (and thus speed-up) is limited by the number of production rules in the system, but the Rete algorithm tends to decrease the level of realizable parallelism significantly. The Rete algorithm only attempts to match rules that have some possibility of being satisfied by the most recent changes to WM. As a result, only processors containing rules that are affected by the most recent changes to working memory perform any useful work during the subsequent production cycle. Measurements by Gupta and Forgy indicate that for most applications, the number of rules affected by a given rule's execution is significantly less than the total number of rules in the application (38:93). Therefore, for a production system using the Rete algorithm, available parallelism is constrained by the number of rules affected by the execution of a given rule. Thus the upper bound speed-up for a given rule execution is equal to the number of rules affected by the execution of that rule. Given that S_i is a subset of the complete rule base (\mathcal{R}) affected by the firing of rule i ,

$$S_i \subset \mathcal{R}$$

then the upper bound is the cardinality of this *affect set*:

$$S_{UB} = |S_i|$$

This equation represents the absolute upper bound on speed-up using production parallelism, however other factors act to significantly reduce this available speed-up.

The pedagogical example in Figure 4.9 illustrates the upper bound restrictions placed on production parallelism by the Rete algorithm; it also shows that using less processors than the size of the affect set can limit the achievable speed-up. For example, executing rule 1 will affect a condition element in the LHS of 3 different rules: 2, 4 and 5. Thus the upper bound speed-up for this particular rule appears to be 3, but the reality of the rule partitioning among three different processors reduces the upper bound speed-up. When rule 1 is executed, only two of the three processors in Figure 4.9 are affected by the changes

Rule Base:

```

(defrule 1 "on P2"      (defrule 2 "on P3"      (defrule 3 "on P2"
  (A ?x)                ?a <- (C $?x)          (A ?x)
  (B ?y)                (E $?x)              (F ?x)
=>                      (F ?y)                =>
  (assert (C ?x)))      =>                  (assert (B ?x))
                        (assert (B ?y))        (assert (D ?x)))
                        (retract ?a))
                        (defrule 4 "on P3"      (defrule 5 "on P1"
  (C ?x)                (C ?x ?y)              ?a <- (B ?x)
  (D ?x ?y)              =>                  (C ?x ?y)
=>                      =>                  (retract ?a)
  (assert (A ?x ?y))      (assert (D ?x ?y))    (assert (F ?y)))
  (assert (B ?y)))      (assert (F ?y)))
  
```

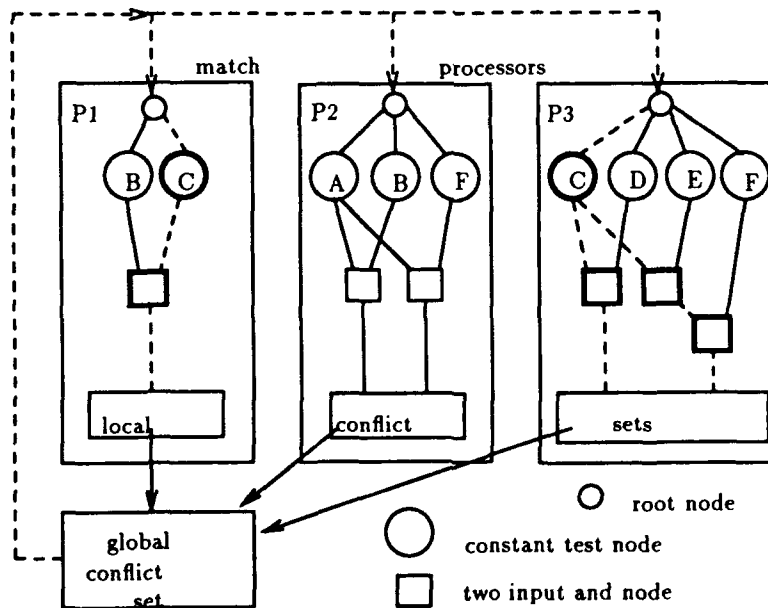


Figure 4.9. Example Ad-hoc Allocation of Rules to Processors

to working memory. The darkened circles and dashed lines in Figure 4.9 trace the path taken by the newly generated Rete network tokens. Although the tokens are sent to all processors, only processors P1 and P3 actually propagate tokens through the Rete network. Therefore the maximum speed-up for executing rule 1 using this rule partitioning is 2. The maximum speed-up for executing a single rule i can be defined as the number of processors containing rules in the affect set, defined as:

$$S_{\max} = N_i$$

where N_i is the number of processors containing rules affected by the firing of rule i . It should be noted here, that the implementation shown in Figure 4.9 does **not** represent an optimum partitioning of rules to processors. Optimum rule partitioning is the subject of the next section. Additionally, the speed-up for this one rule does not reflect the available parallelism for the complete application. The concept of maximum parallelism and average parallelism is discussed in Section 4.5.4.

The example in Figure 4.9 also illustrates another significant problem associated with production parallelism: the loss of Rete network sharing (36:47). One of the advantages of the Rete algorithm is the sharing of common condition elements between similar rules. When production rules are hosted on different processors, the network sharing concept no longer yields the level of sharing available in the sequential implementation; different processors may possess the same combinations of condition elements, but these condition elements are no longer shared. The loss in Rete network sharing introduces overhead in the form of redundant computation. For example, in Figure 4.9, executing rule 1 causes both processor P1 and processor P3 to perform "C" class constant tests instead of the one "C" class constant test that would be required by a sequential implementation. Despite the loss in sharing among the constant test nodes, the number of two-input-and nodes in the parallel implementation is the same as the number of two-input-and nodes in the sequential implementation. The loss in speed-up due to losses in Rete network sharing is proportional to the ratio of affected nodes in the sequential implementation to the number of affected nodes in the parallel implementation (36:47). The nominal speed-up resulting

from losses in Rete network sharing can be expressed as:

$$S_{\text{nominal}} = S_{\text{max}} \cdot \frac{R_{\text{seq}}}{R_{\text{par}}}$$

where R_{par} is the number of Rete nodes in the parallel implementation and R_{seq} is the number of Rete nodes in the sequential implementation. However, because the time to update two-input-and nodes is typically much greater than the time required to update constant test nodes, this formula applies mostly to the ratios of two-input-and nodes between the parallel and sequential versions (36:47). In the case of the example in Figure 4.9, the ratio of R_{par} to R_{seq} is 1, therefore $S_{\text{nominal}} = S_{\text{max}}$.

Finally, one of the most significant problems in using production parallelism concerns the effective balancing of the working memory update task among the involved processors. If the working memory updates for different partitions (processors) are not equal, available speed-up will be reduced by load imbalance. For a given production cycle, speed-up is limited by the match processor requiring the longest time to update the state of its local Rete network; therefore the available speed-up is proportional to the ratio of average time required by all match processors to the maximum time required by all match processors (36:47). That is, the true speed-up is constrained by:

$$S_{\text{true}} = \frac{\frac{\sum_{i=1}^P t_i}{P}}{\text{MAX}_{i=1}^P t_i} \cdot S_{\text{nominal}}$$

where P is the number of processors and t_i is the time required for processor i to update its local Rete network. Measurements on expert systems indicate that load imbalance among processors reduces the average speed-up by approximately 5 times in the average application (36:47). Unfortunately, the times required to update the local Rete networks can be obtained only through high resolution run-time measurement (59:95). As a result, this information cannot be used to to make a static determination of how task update time affects the potential for speed-up.

4.5.3 Deriving an Optimum Rule Partition Researchers have taken many paths in attempting to obtain an “optimum” partitioning of rules to processors (59) (48) (72) (22), but in the context of this problem, the concept of optimality is difficult to capture. Clearly, an optimum rule partitioning is the one that consistently produces the largest speed-ups, but how is this rule partitioning derived? Most methods reviewed rely on static analysis of the rule base or arbitrary methods to partition the rule base among available processors. Static analysis using information derived from actual application execution traces can be combined with simulated annealing in an effort to optimize the rule partitioning for specific applications “runs” (59:97). However, as Figure 4.10 shows, even the simulated annealing approach falls far short of the upper-bound speed-up because of the variations in processing times among the rule partitions on different processors. Additionally, a simple *round-robin* (RR) partitioning method demonstrated by Graham (34:419) falls within 4% of the optimum partitioning (59:97). Research clearly shows that a close to optimum partitioning can be achieved using static methods, however the RR method does not consider the context sensitive nature of rule partitioning¹ (59:97). In order to perform an evaluation of the available parallelism, a static and context-sensitive rule partitioning method is required.

The ideal partitioning of a given rule base would cause each processor to be evenly affected (requires the same amount of time to update its local Rete network) by any given rule execution. Since it is impossible to determine the time required to perform the update cycle resulting from a given rule execution using static methods (59:96), a somewhat less constrained approach is required. An earlier study demonstrates a method whereby a rule base could be partitioned so that a given rule firing on one processor would affect the greatest number of other processors (67). This method does not guarantee that the local Rete network update times on each affected processor are equal, but it allows the largest possible number of processors to perform useful work during each production cycle. The

¹In other words, the ordering of the rules in the rule base input file will affect the partitioning, an undesirable characteristic

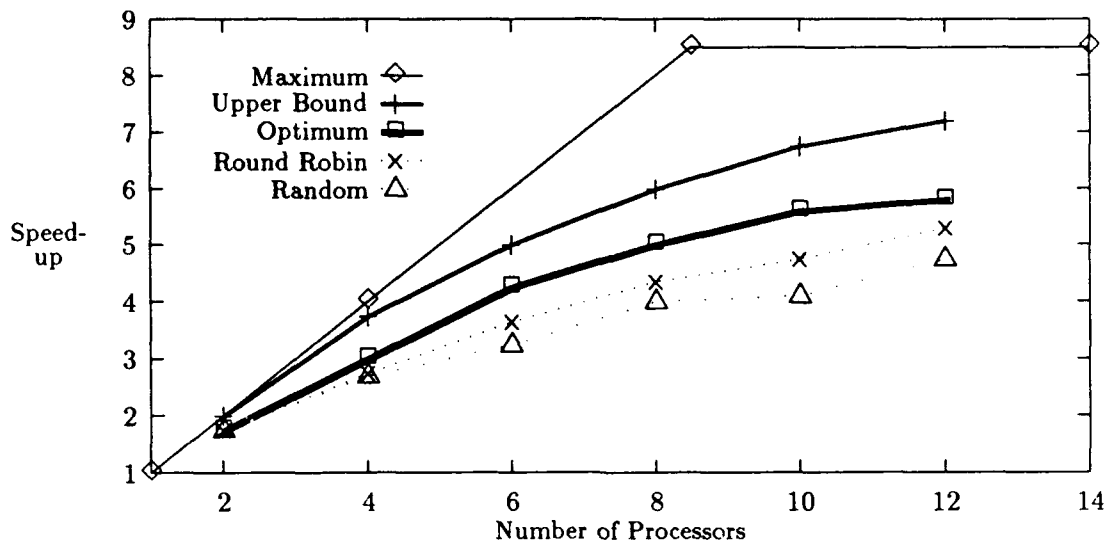


Figure 4.10. Selected Rule Partitioning Results From Oflazer's Research (59:98)

problem of maximizing the "span-of-effect" of each rule execution can be stated as (67:18):

Maximize:

$$\sum_{i=1}^R \sum_{j=1}^N \sum_{k=1}^R p_{i,j} p_{k,j} S_{k,i} \quad \text{the span of affect}$$

Subject to:

$$\sum_{j=1}^N p_{i,j} = 1 \quad 1 \leq i \leq R. \quad \text{each rule assigned to 1 processor}$$

Where:

$$p_{i,j} = \begin{cases} 1 & \text{if rule } j \text{ assigned to processor } i \\ 0 & \text{otherwise} \end{cases}$$

and:

$$S_{i,j} = \begin{cases} 1 & \text{if executing rule } j \text{ affects rule } i \\ 0 & \text{otherwise} \end{cases}$$

The result of the "maximize" term is the total number of rule partitions affected by the rule to processor assignments as defined by the matrix $p_{i,j}$ and the characteristics of the

rule base defined by the matrix $S_{i,j}$. This equation is developed in more detail in Appendix C. The total number of rules in the rule base is R and the total number of processors is N . Examples of the \mathcal{P} and \mathcal{S} matrices for the partitioning shown in Figure 4.9 are shown in Figure 4.11. Using this method, the rules in Figure 4.9 can be repartitioned as shown in Figure 4.12 so that the firing of any rule now affects at least one rule on each of the three processors. However, the above described rule-to-processor partitioning process is shown to be *NP-Complete*:

In order to show that a given problem is *NP-Complete*, we must show that it can be solved in polynomial time on a Non-Deterministic Turing Machine and that a known *NP-Complete* problem can be mapped to this problem in polynomial time. In order to find the optimal solution to this problem, every possible allocation of rules to processors must be evaluated either explicitly or implicitly. The simple case of allocating a set of n rules to 2 processors is sufficient to show that the problem has exponential time complexity.

Any set \mathcal{Z} of n rules comprises 2^n possible subsets:

$$\mathcal{N} = 2^{|\mathcal{Z}|}.$$

Under the constraint of the problem, assigning a subset of the elements in \mathcal{Z} to one processor requires that the remainder be assigned to the other processor:

$$\mathcal{X} \subset \mathcal{Z}$$

$$\mathcal{X} \rightarrow P_1$$

$$\mathcal{Z} - \mathcal{X} \rightarrow P_2.$$

From, this example, it is obvious that 2^n possible assignments of rules to processors exist, each of which must be considered. On the surface, it might appear that adding additional processors and rules will increase the complexity of the problem to:

$$\mathcal{N} = \mathcal{P}^{|\mathcal{Z}|}$$

where \mathcal{P} is the number of processors and \mathcal{Z} is the set of rules yielding \mathcal{N} possible assignments of rules to processors. However, by making one simplifying assumption, the problem can be reduced to a 0-1 integer linear programming (ILP) problem: The assumption is that in symmetrical states, the communication time between all processors is approximately equal. Considering the advanced message passing techniques and memory access techniques of current generation multi-computers, this is not a rash assumption. The author's own experience plus several other papers and sources of technical information

tend to support this assumption. Expressing the communication cost between processors i and j as $C_{i,j}$, the assumption can be expressed as:

$$\{\forall i \forall j \in \mathcal{P} | C_{i,j} = C_{const}\}.$$

Assuming the communication costs are approximately equal, symmetrical assignments of rules to processors become equivalent. Using the two processor example this can be expressed as:

$$\mathcal{X} \subset \mathcal{Z}$$

$$Cost(\mathcal{X} \rightarrow P_1, \mathcal{Z} - \mathcal{X} \rightarrow P_2) = Cost(\mathcal{Z} - \mathcal{X} \rightarrow P_1, \mathcal{X} \rightarrow P_2).$$

Thus, the cost of inter-processor communication does not affect the time complexity of the rule-to-processor partitioning. Assuming the existence of a Non-Deterministic Turing Machine (NDTM) with at least 2^n processing elements² and knowing that the evaluation of each possible state requires polynomial time (from the span-of-affect equation), then the optimum solution can be found in polynomial time. This is sufficient for proving the first requirement for *NP-Completeness*.

For the second part of the *NP-Completeness* proof, we need to find a known *NP-Complete* problem and show a mapping from that problem to the rule-to-processor allocation problem. The Assignment problem is a well known *NP-Complete* problem that attempts to minimize the cost of assigning n resources to n requesters. This problem is described as (46:253):

Minimize:

$$\sum_{j=1}^n \sum_{i=1}^n c_{i,j} x_{i,j}$$

Subject to:

$$\sum_{i=1}^n x_{i,j} = 1 \quad 1 \leq j \leq n$$

$$\sum_{j=1}^n x_{i,j} = 1 \quad 1 \leq i \leq n.$$

By mapping the n resources to the n production rules and the n requesters to the n processors of our n rule to n processor allocation problem, it appears, that the allocation problem can be solved almost directly. In fact, the assignment problem turns out to be a subset of the rule to processor allocation problem because future resource to requester assignments are not dependent on the resources already assigned. By replacing the Minimize term of the original equation with the minimize term of the assignment problem still leaves the

²A true NDTM has an infinite number of processing elements.

$$\mathcal{P} = \begin{vmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{vmatrix} \quad \mathcal{S} = \begin{vmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{vmatrix}$$

Figure 4.11. Rule-Processor Allocation Table and Rule-Antecedent Cross Reference Table

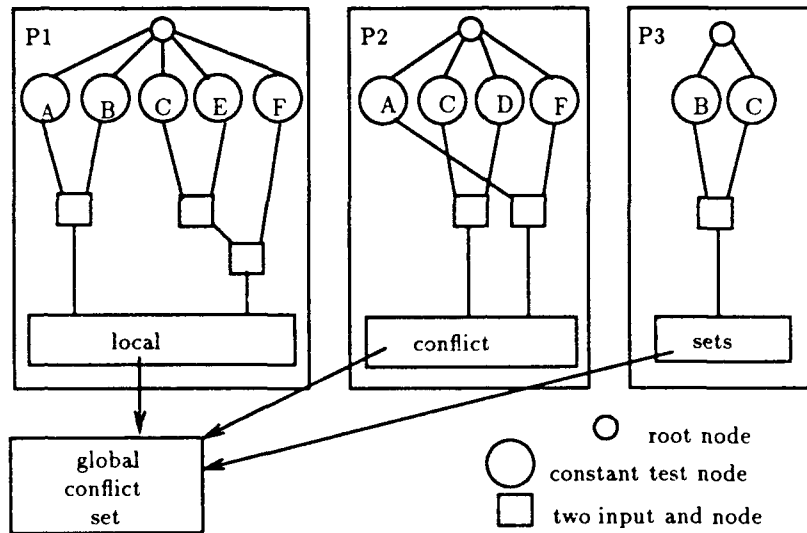


Figure 4.12. Optimum Partitioning of the Rule Base in Figure 4.9 for 3 Processors

problem of non-compatible problem constraints. Considering the case of an non-symmetrical assignment problem (n requesters, r resources), this problem no longer exists. These types of problems are very common and their constraints reduce to the requirement of having at least one resource assigned to one requester which matches the rule to processor allocation problem (46:258). Demonstrating this ad-hoc mapping of the two problems fulfills the second part of the requirement for proving the *NP-Completeness* of the problem.

If the average affect set size is much smaller than the number of processors, then there may be a number of different rule partitionings that maximize the span-of-affect. In the case where more than one maximum span-of-affect rule partition exists, a secondary method for choosing which of the maximum span-of-affect partitions provides the best potential for increasing speed-up may be useful. One possible selection method attempts

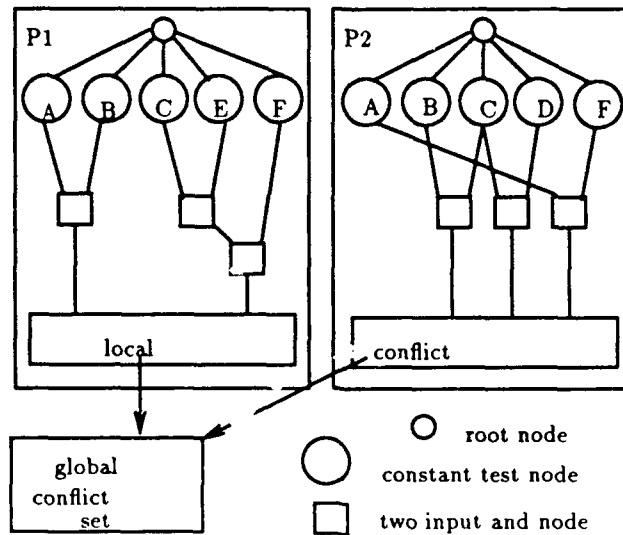


Figure 4.13. Optimum Partitioning of the Rule Base in Figure 4.9 for 2 Processors

to evenly distribute the number of two input Rete nodes that are potentially affected by a given rule activation. For example, given the rule base in Figure 4.9, the rules can be partitioned between 2 processors in seven different ways that maximizes the span-of-affect³. Out of the seven possible maximum span-of-affect partitionings, two possess the same minimum difference in the number of two input Rete node updates. Due to the unpredictable nature of run-time data flow through the Rete networks, the static “minimum difference partitioning” does not guarantee to minimize the processing difference between processors. Although the secondary partitioning method does not guarantee optimum load balance among the available processors, it appears to make an optimum load balance situation more likely.

4.5.4 Measuring Production Parallelism Now that the important characteristics of production parallelism have been explored, it is possible to use some of these characteristics to roughly quantify available speed-up using production parallelism. As noted in the previous section, several of the factors adversely affecting production parallelism cannot be used for a static determination of available parallelism because they depend on the

³There are $2^N/2 = 16$ possible ways of partitioning this rule set among two processors

dynamic characteristics of the system. Information concerning rule execution affect-sets is available directly from static analysis. However, information concerning the differences between the average and maximum processing times and the number of Rete nodes affected by a given rule firing require dynamic measurement. Because 2 out of the 3 factors affecting parallelism cannot be accounted for using static analysis, the actual measures presented in this section must represent an **approximate upper bound** on available speed-up when using production parallelism.

A rough upper bound on parallelism is simply the average number of rules affected by the firing of a given rule in the rule base:

$$S_{UB} = \frac{\sum_{i=1}^R \sum_{j=1}^R S_{i,j}}{R}$$

However, this rough bound assumes that each rule is executed an even number of times, a condition that is often not true. If the number of times a given rule i is executed in proportion to the number of rules that are actually executed, then a more accurate determination of available parallelism is:

$$S_{UB} = \sum_{i=1}^R w_i \cdot \sum_{j=1}^R S_{i,j}$$

where

$$w_i = \frac{\text{Number of time rule } i \text{ executed}}{\text{Total number of rules executed}}$$

For a given rule, w_i cannot typically be determined with any accuracy using only surface characteristics; information from actual application execution is required. This type of information is readily available from both CLIPS and OPS-5 shells through the use of "watch" commands (18:61)(27:48).

The cardinality of the affect set provides an absolute upper bound on the potential for speed-up when no rule partitioning is specified, but production parallelism relies on static partitioning which is likely to adversely affect achievable speed-up. If the rule base is partitioned among a set of N processors, a tighter bound of available parallelism becomes

available. The upper bound now becomes the average number of processors containing a rule that is affected by the execution of another rule:

$$S_{UB} = \frac{\sum_{i=1}^R \sum_{j=1}^N \sum_{k=1}^R \mathcal{P}_{i,j} \mathcal{P}_{k,j} S_{k,i}}{N}$$

This upper bound figure again assumes that each rule will be executed an even number of times. To obtain a speed-up figure that is more dependent on the actual proportion of rule execution in a typical application, the following formula can be used:

$$S_{UB} = \sum_{i=1}^R w_i \cdot \sum_{j=1}^N \sum_{k=1}^R \mathcal{P}_{i,j} \mathcal{P}_{k,j} S_{k,i}$$

where w_i is defined in the previous paragraph as the proportion of times rule i is executed to the total number of rules executed. This final formulation provides some insight into how the static partitioning of the rule base affect available speed-up.

4.5.5 Affect of Agenda Parallelism on Production Parallelism Agenda parallelism provides the opportunity to significantly increase production parallelism through execution of multiple rules during the same production cycle. Multiple rule execution (if more than one rule is actually executed) results in increasing the number of rules affected and/or increasing the granularity of the Rete network update task on one or more processors. When instantiations of **different** rules are executed in parallel, then the first effect (an increase in the number of rules affected) is likely to result. Executing instantiations of different rules is desirable because the size of the affect set, and thus the number of processors performing useful work increases. Executing instantiations of the same rule is likely to be less desirable because it does not increase the size of the affect set and can result in significant load imbalance among the available processors.

4.6 Node Parallelism

Node parallelism is a finer grain level of parallelism that attempts to achieve speed-ups through the processing of different nodes of the Rete network in parallel. As a result,

this decomposition is specific to the Rete algorithm and cannot be generalized for other production system processing approaches (36:48). In the node parallelism approach, one global Rete network takes the place of all the local Rete networks used in the production parallelism approach. Multiple processors can then be used to process Rete node activations in parallel as working memory tokens flow through the network. The two primary reasons for using node parallelism are (36:49):

1. It has the potential to significantly reduce the maximum time required to process changes to working memory, a factor that is shown to have a significant negative impact on production parallelism (36:47).
2. It "reclaims the loss in Rete network sharing experienced by production parallelism" (36:49). This second factor is not as significant as the first one, but does have some impact (36:47).

On the negative side, node parallelism carries the penalty of increased communication costs. When using node parallelism, this communication cost is proportional to the number of successful Rete node activations processed in a given cycle (36:49). The increased cost in communications of node parallelism over production parallelism appears to significantly limit the parallel architectures that can efficiently implement node parallelism.

4.6.1 Approaches to Implementing Node Parallelism Several different methods for implementing node parallelism have emerged from past research at Carnegie-Mellon University (CMU) (36) (40)(39)(75), Columbia University(CU) (44)(72), and Merit Technologies (51). Each of these approaches have shown ten-fold speed-up through simulation or actual implementation. The demonstrated performance of each of these approaches marks them as likely candidates for increasing the execution speed of many production system applications.

The node parallelism approach taken by researchers at CMU concentrates only on the most compute intensive part of the production cycle, the match phase, and effectively ignores other potential sources of parallelism. The *task-level* approach involves using a set

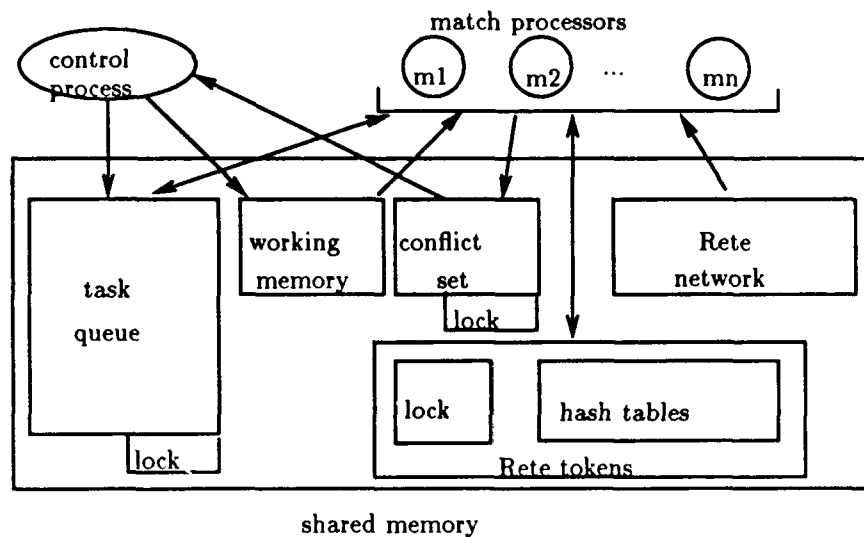


Figure 4.14. The CMU Production System Machine Implementation (40:106)

of *match processors* to process two-input Rete node⁴ activations in parallel. The list of node activations to be processed is maintained on a central queue and processors dequeue and enqueue the two-input node activation tasks (40:102). Abstractly, the processing of two-input activations is similar to the *worker-controller* approach used to solve the N-queens problem in Chapter 3, Section 3.1.2. Other parts of the production cycle, conflict resolution, single input node activations and RHS actions, are performed sequentially by a single processor (40:102). Figure 4.14 shows the system level layout of the CMU implementation of Node level parallelism on a 16 processor Encore Multimax multi-processor.

In contrast to processing two input node activations in parallel, researchers at Merit Technology concentrate primarily upon performing parallel processing of constant-test (single input) nodes in parallel. The METE system accomplishes the match phase update to the Rete network in two phases: the Constant-Test (ct) phase and the Two-Input-And (TAND) phase. In the first (ct) phase, a number of ct processors perform single tests on tokens in parallel and forward satisfied tokens to different (sometimes more than one) TAND processor(s). The TAND processors queue up the satisfied productions sent to them until all constant tests have been completed. When all constant tests have been processed,

⁴The two types of two-input Rete nodes are: *and* nodes and *not* nodes.

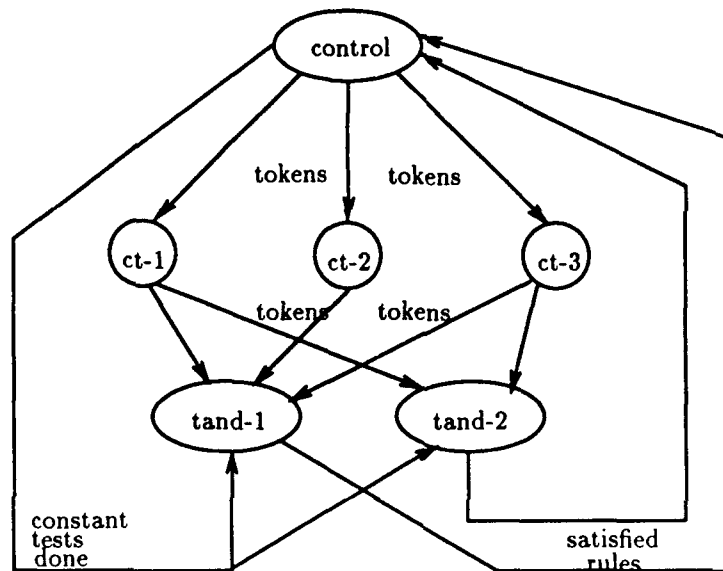


Figure 4.15. The METE Algorithm Parallel Inference Architecture (51:103)

each TAND processor with tokens executes all two-input Rete node activations required to satisfy a given rule. A given TAND processor may contain several two input Node tests depending on the attributes of the rule base (51:102)(50). Figure 4.15 shows the operation of the METE algorithm for updating a simple two rule application. In this figure, each “node” represents a single processor and each arc represents interprocessor communication via shared memory. During the ct phase, the METE algorithm uses node parallelism, but switches to production parallelism in the tand phase.

Researchers at CU take much the same approach as Merit Technologies except that the algorithm is hosted on a massively parallel, heterogeneous architecture known as NON-VON (Figure 4.16). The NON-VON approach uses node parallelism in the constant test phase and production parallelism in the two-input test phase. Prior to execution, the constant-test nodes are distributed among the small processing elements and the two-input nodes are distributed among the large processing elements. Like the METE algorithm, execution is essentially in four phases instead of the typical three as there is a distinct break in the match phase. During the first part of the match phase, all small processing elements evaluate the constant test nodes in parallel within a few machine cycles and pass

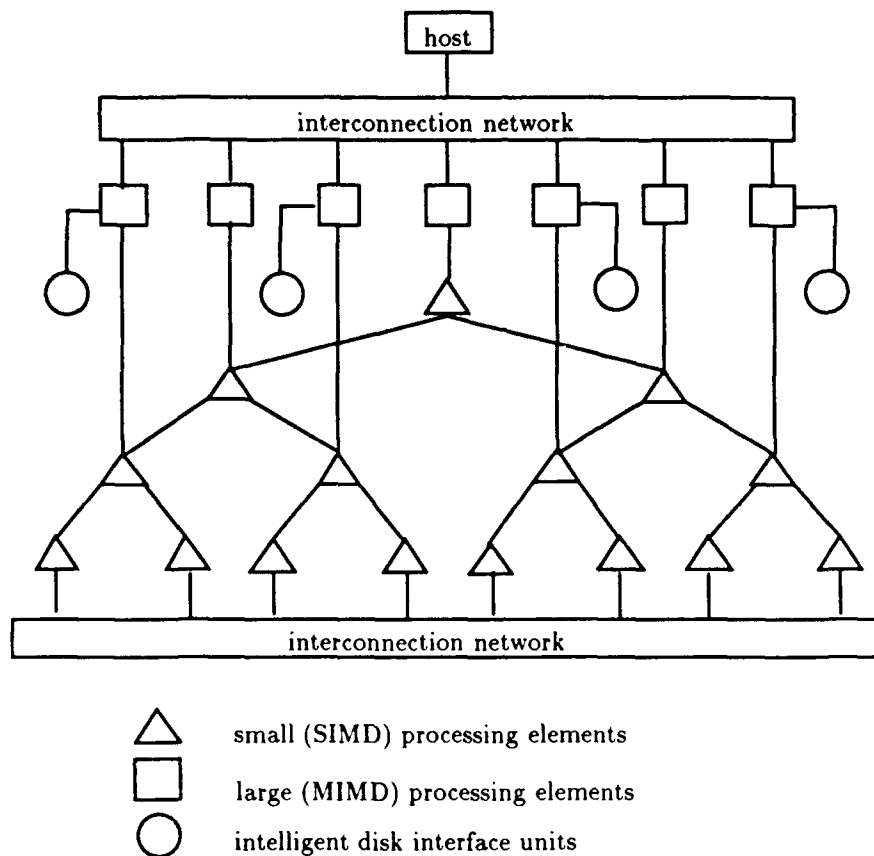


Figure 4.16. Organization of the NON-VON Machine (44:242)

satisfied tokens to the large processing elements. When the small processing elements have completed the constant test processing, the large processing elements can perform the two-input node tests. The CU approach ignores most of the potential for parallelism in the select and act phases as the host processor performs both the selection of the next rule to executes many of the resulting RHS actions (44:246).

In a proposed multi-computer implementation of the CMU approach, a two phase approach possessing attributes of both the original CMU implementation and the METE algorithm is proposed. The major difference in this third approach is that the operation of TAND processors is fully pipelined with the operation of the constant test processors instead of requiring the TAND processors to wait until constant test processing is completed. In the proposed mapping, the control processor begins the cycle by sending tokens

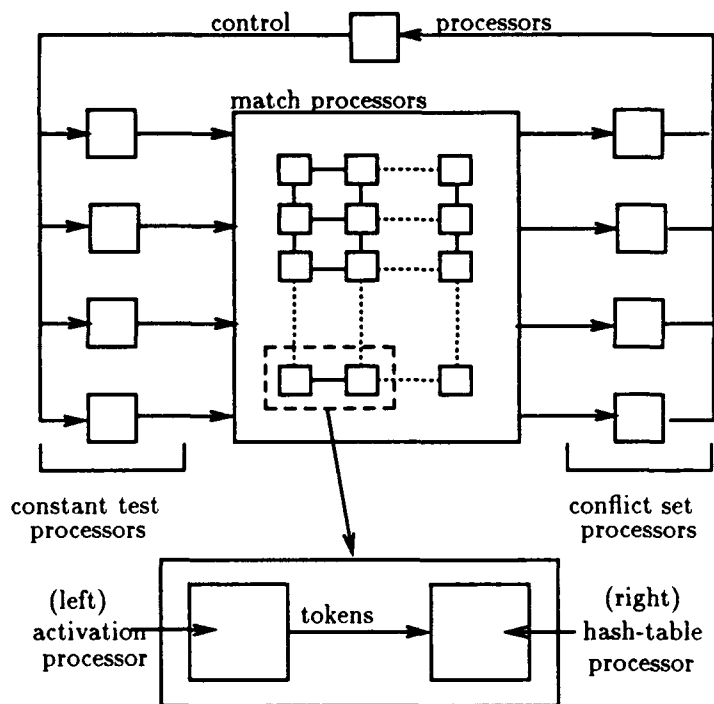


Figure 4.17. High-Level View of Proposed Multi-Computer Mapping (75:7)

generated by the previous action phase to the constant test processor. The constant-test processors pass satisfied tokens to selected match processors based on a distributed hash table implementation. The match processors work in pairs to perform two-input node tests and pass satisfied tokens to other processors or the conflict set processors (if a rule is fully satisfied). Finally, the conflict set processors work together to determine the next rule to execute (75:5-6). If the number of match processors is small (32 or less), then this multi-processor model proposes that the control processor be used to perform constant tests and conflict resolution (75:10). Figure 4.17 shows a high-level view of the proposed multi-computer mapping.

4.6.2 An Overall Model for Node Parallelism All four of the approaches reviewed in the previous section possess very different characteristics and compromise one or more of the important aspects of “pure” node parallelism. Of these approaches, a hybrid of CMU’s two approaches (40)(75) appears to best preserve the concepts of node parallelism

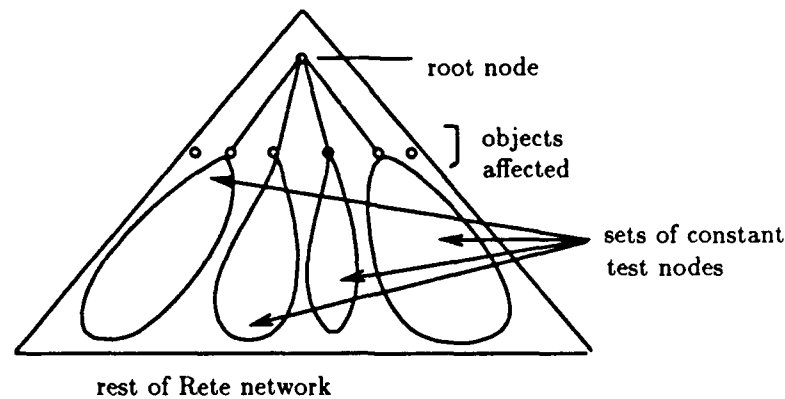


Figure 4.18. Scheduling Constant Test Nodes (36:76)

while maintaining the requisite maximum efficiency. Both of CMU's approaches use a sequential evaluation of the constant-test nodes and parallel evaluation of the two-input-test phase. Data taken from a number of production system application measurements shows that two-input-tests account for 75 to 95% of the total Rete state update time while the constant-test require a much smaller percentage (5 to 25%) of the total Rete network update time. This means that the CMU approach uses true node parallelism 75 to 95% of the time. Additionally, it is very possible to modify the current CMU approach to incorporate parallel evaluation of the constant-test nodes (36:76). Conversely, the CU and METE approaches use node parallelism only 5 to 25% of the time and rely substantially on production parallelism for evaluating the two-input nodes. The CU and METE approaches cannot be easily modified for more complete node parallelism, therefore these approaches are better covered by the extensive discussion of production parallelism in Section 4.5.

The proposed overall model for node parallelism proposes limited parallelism in evaluating constant-test nodes. Using an approach suggested by Gupta, clumps of constant test nodes are evaluated together instead of as single node activations (36:76). Clumps, as opposed to single nodes, are evaluated in parallel because the task of evaluating one constant test node is very small compared to the time required to obtain the task from a queue and schedule any subsequent node activations (36:75). This investigation chooses to "clump" the constant-test nodes by class type (first field of an LHS pattern) because it allows for relatively easy division of the nodes to be tested using an indexing technique

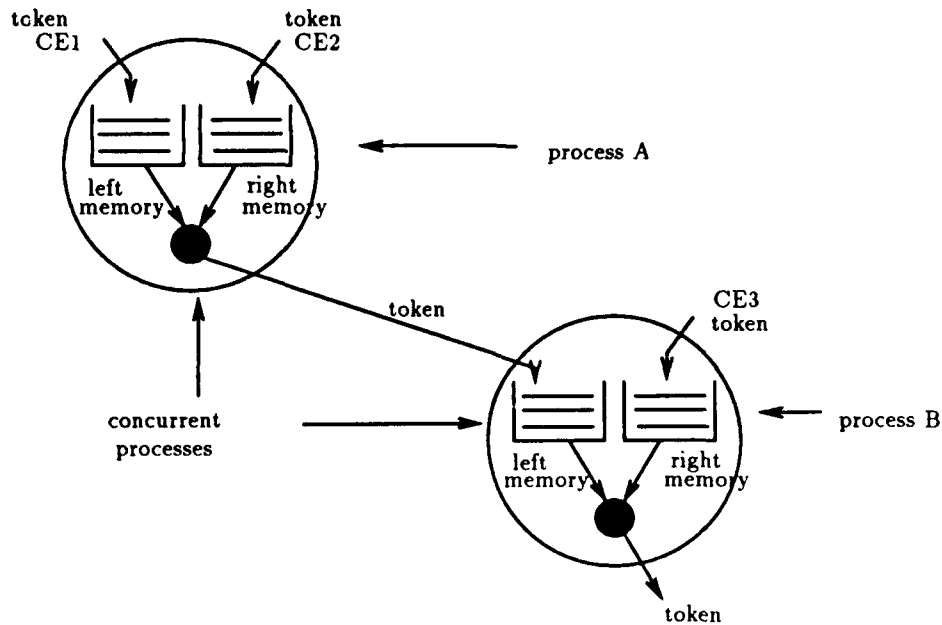


Figure 4.19. A Parallel Two-Input Node Activation (36:49)

such as hashing. Additionally, dividing the constant-test nodes by class allows a very high-level division of the constant-test nodes into tasks that may be closer in computational requirements to the task of updating a single two-input Rete node⁵. Finally, dividing the constant-test nodes allows the use of multiple processors to queue up new tasks, an advantage that becomes evident later.

The second phase of the node parallelism match process, two-input node activations is taken directly from the current CMU implementation. In this approach, the task of updating a single two-input node represents a separate, independently executable process as shown in Figure 4.19. Although this node parallelism approach appears straightforward, there are several significant differences between a sequential and parallel node activation. Each node should possess local right (alpha or beta) and left (alpha) memory nodes instead of using global memory nodes as in the sequential Rete network; if each process did not possess local memory nodes, then all two-input activations would be required to take place in lockstep (e.g. match process B in Figure 4.19 could not proceed until match process

⁵Recall from Section 4.5 that two-input nodes requires, on the average, approximately 20 times more processing time than a single-input node

A completes and the process providing the CE3 input also completes) (36:49). Another important aspect of node parallelism is that only certain types of node activations are concurrently processable. Only right or left inputs to a given process can be processed at the same time, however multiple tokens input to the same side can be processed concurrently (36:49). In Figure 4.19, if token CE1 and CE2 reach process A at approximately the same time, only one of the two tokens can be processed immediately, the other must wait. The condition of having inputs to different sides of the same two-input node is referred to as a "token collision".

4.6.3 Measuring Node Parallelism Node parallelism has the potential for greater amounts of concurrency than production parallelism, but the bounds on node parallelism are almost as difficult to quantify. Using the assumptions outlined in the preceding section, the bounds on node parallelism must be quantified in two parts: the constant-test node parallelism and the two-input node parallelism. The level of constant-test node parallelism is simply the number of objects affected by the most recent rule execution. In terms of two-input Rete nodes, node parallelism is limited to the number of two-input nodes in the Rete network that are directly affected by the most recent changes to working memory. Theoretically, all nodes in the Rete network could be affected by the most recent changes to working memory, but measurements by Gupta and Forgy assure us that this is not generally the case (38). Whenever the most recent changes to working memory result in a token collision, the number two-input Rete nodes that can be processed concurrently is reduced by one. The concept of Rete node activations and token collision may not be entirely clear from the textual description, therefore a specific example is provided.

The global Rete network in Figure 4.20 illustrates how an upper bound on node parallelism can be established for the sample application first introduced in Figure 4.9. Consider the particular case of executing rule 2: LHS patterns associated with objects "B" and "C" are directly affected by the execution of predicates of the RHS of rule 2, therefore the constant test nodes associated with these objects become the basis for parallel node evaluation. There are only 2 constant test objects associated with the RHS of rule 2, therefore the extent of constant-test parallelism is 2. Object pattern B has 2 direct

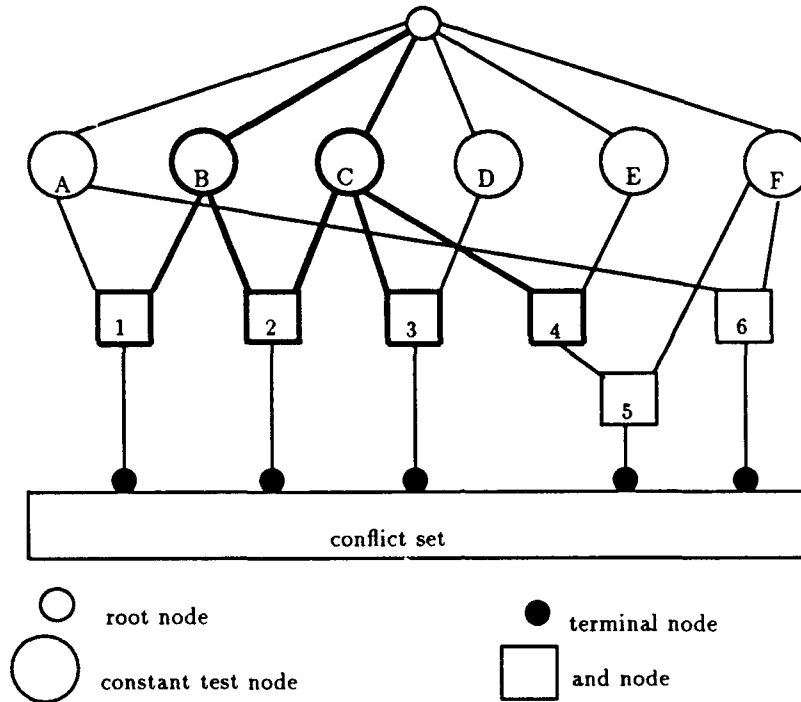


Figure 4.20. An Example for Measuring Node Parallelism

successors in the Rete network and object pattern C has 3 direct successors in the Rete network, therefore the maximum extent of two-input node parallelism is 5. However, one of the five affected nodes (number 2) is affected directly by both object pattern B and object pattern C resulting in a token collision. The token collision reduces the number of two-input Rete nodes that can be processed concurrently to 4.

Establishing a bound on node parallelism is similar in many ways to the method used to bound production parallelism, however since the Rete network is considered global the additional problem of the most effective way to partition the rule base is not a factor. The actual measurement of node parallelism consists of a weighted combination of the parallelism in the two different parts of the match phase:

$$S_{UB} = (W_c \cdot P_c) + (W_t \cdot P_t)$$

which means that speedup is equal to the proportion of time spent in the constant-test

phase (c) times the amount of parallelism in the constant-test phase, plus the proportion of time spent in the two-input test phase (t) times the parallelism in the two-input test phase. Weighting factors for W_c fall between 0.05 and 0.25 and the weighting factors for W_t fall between 0.75 and 0.95 (36:45). Because of the assumptions that have been made (primarily the clumping of affected patterns in the constant-test phase), the level of parallelism in the two-input test phase is always greater than or equal to the parallelism in the constant-test phase. Therefore, the maximum node parallelism occurs when $W_c = 0.05$ and $W_t = 0.95$ and the minimum node parallelism occurs when $W_c = 0.25$ and $W_t = 0.75$. In order to quantify the level of node parallelism, all that remains is to quantify each of the P terms for the above equation.

It is possible to quantify node parallelism using the ILP terms like those used to solve the rule partitioning problem and to quantify the level of production parallelism in Section 4.5, however quantifying node parallelism is not an ILP problem. To determine the level of node parallelism in a given rule base, the three matrices, \mathcal{A} , \mathcal{B} and \mathcal{C} are used to define the salient features of the rule base. These matrices are defined as follows:

Affect Patterns:

$$\mathcal{A}_{i,j} = \begin{cases} 1 & \text{if executing rule } j \text{ affects pattern } i \\ 0 & \text{otherwise} \end{cases}$$

Contains Patterns:

$$\mathcal{B}_{i,j} = \begin{cases} 1 & \text{if rule } j \text{ contains pattern } i \\ 0 & \text{otherwise} \end{cases}$$

Colliding Patterns:

$$\mathcal{C}_{i,j} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are the first two} \\ & \text{patterns in any rule} \\ 0 & \text{otherwise} \end{cases}$$

As seen later, each of these matrices plays a critical part in the determination of bf upper-bound node parallelism.

Before the \mathcal{A} , \mathcal{B} and \mathcal{C} matrices for the example problem in Figure 4.9 can be properly defined, a set of all unique LHS patterns must be constructed. For the example problem,

the patterns, in order of appearance, are:

1. (A ?)
2. (B ?)
3. (C ?)
4. (E \$?)
5. (F ?)
6. (D ? ?)

Prior to constructing the \mathcal{A} matrix representation of the example rule base, the RHS patterns affected by each rule need to be stated:

1. rule 1 RHS affects pattern (C ?)
2. rule 2 RHS affects patterns (B ?) and (C ?)
3. rule 3 RHS affects patterns (B ?) and (D ? ?)
4. rule 4 RHS affects patterns (A ?) and (B ?)
5. rule 5 RHS affects patterns (B ?), (D ? ?) and (F ?)

From the RHS patterns affected and the pattern set representation above the \mathcal{A} matrix for the example problem can be defined as:

$\mathcal{A} =$

		rule				
		1	2	3	4	5
pattern	1	0	0	0	1	0
	2	0	1	1	1	1
	3	1	1	0	0	0
	4	0	0	0	0	0
	5	0	0	0	0	1
	6	0	0	1	0	1

For instance, rule 1 affect only one patterns type, (C ?), which is pattern number 3 in the pattern set, thus $A_{1,3} = 1$. It is assumed that unless a given rule explicitly affect a given pattern, the matrix location is 0!

The set of possible LHS patterns have already been defined; however, before the initial B matrix can be constructed, the patterns contained by a given rule must be known. For the example rule base, the LHS patterns are:

1. rule 1 contains patterns (A ?) and (B ?)
2. rule 2 contains patterns (C ?), (E \$?) and (F ?)
3. rule 3 contains patterns (A ?) and (F ?)
4. rule 4 contains patterns (C ?) and (D ? ?)
5. rule 5 contains patterns (B ?) and (C ?)

From the LHS patterns contained in a given rule and the set of possible LHS patterns, the following B matrix can be constructed for the example problem:

$B =$

		rule				
		1	2	3	4	5
pattern	1	1	0	1	0	0
	2	1	0	0	0	1
	3	0	1	0	1	1
	4	0	1	0	0	0
	5	0	1	1	0	0
	6	0	0	0	1	0

For example, rule 1 contains patterns (A ?) and (B ?) which correspond to patterns 1 and 2 of the pattern set respectively. Therefore, the locations of the B matrix, $B_{1,1}$ and $B_{1,2}$, take on the value 1. Again, it is assumed that $B_{i,j} = 0$ unless rule i contains pattern j . In the simple example problem (Figure 4.9), no two input Rete node sharing is used because none of the rules have more than one pattern in common, thus the B matrix is accurate as it is written above. However, in a more complex problem, the B matrix must be "reduced"

so that it accurately represents the number of two-input nodes in the Rete network. The "reduction" algorithm is discussed further in Chapter 5.

In constructing the final (C) matrix, the set of LHS patterns and a sub-set of the information used for construct the B matrix is required. Specifically, the first two patterns that make up the LHS of each production rule must be known. For the example problem, the first two patterns for each production rule are as follows:

1. for rule 1 (A ?) and (B ?) are the first two patterns
2. for rule 2 (C ?), (E \$?) are the first two patterns
3. for rule 3 (A ?) and (F ?) are the first two patterns
4. for rule 4 (C ?) and (D ? ?) are the first two patterns
5. for rule 5 (B ?) and (C ?) are the first two patterns

Using the above first-two-pattern information and the information contained in the pattern set, the C matrix for the example problem is:

		pattern					
		1	2	3	4	5	6
$C =$	1	0	1	0	0	1	0
	2	1	0	1	0	0	0
	3	0	1	0	1	0	1
	4	0	0	1	0	0	0
	5	1	0	0	0	0	0
	6	0	0	1	0	0	0

In the case of the above matrix, rule 1 contains patterns (A ?) and (B ?) as the first two patterns which correspond to patterns number 1 and 2 of the pattern set. Therefore, the C matrix locations $C_{1,2}$ and $C_{2,1}$ are equal to 1 indicating a pattern conflict exists. Again, it is assumed that unless two patterns i and j form the first two patterns of a rule then $C_{i,j} = 0$ and $C_{j,i} = 0$

From the model of node parallelism discussed in the previous section, the number of constant test nodes is simply the number of different pattern classes affected by a given rule execution. This is equivalent to the number of patterns affected by the execution of a given rule i which is the same as summing the contents of a given column i of the \mathcal{A} matrix representation. Using the \mathcal{A} matrix representation for the example problem, the amount of node parallelism in the constant-test phase can be defined mathematically as:

$$P_c = \frac{\sum_{i=1}^R \sum_{j=1}^P \mathcal{A}_{i,j}}{R}$$

where R is the number of rules and P is the number of patterns.

The level of two-input node parallelism is the average number of Rete nodes affected by a given rule execution minus the average number of Rete token collisions divided by two (only one of the tokens can be processed immediately; the other can not). To find the number of Rete node tokens affected by the execution of a given rule i , it is necessary to determine for each pattern j that exist on the RHS of rule i , how many rules k contain a pattern that is affected by that pattern (j). The sum of all the Rete node patterns (ks) represents the level two-input-node parallelism (not accounting for possible token collisions) for the rule i . This process can be defined by the following pseudo-code fragment:

1. for a given column i (representing a rule) in the \mathcal{A} matrix:
 - (a) if the j th item is a 1 then
 - i. locate the corresponding row j in the \mathcal{B} matrix and sum the number of patterns in that row. This is the total number of rules k that contain a patterns that is affected by pattern j and
 - ii. add this value k to the total number of affected patterns.
 - (b) else if the value of the of the j th item is a 0, then do nothing.

Over the total rule base of R rules, the total number of two-input Rete nodes affected can be defined mathematically as:

$$P_{2-in} = \sum_{i=1}^R \sum_{j=1}^P \sum_{k=1}^R \mathcal{A}_{i,j} \mathcal{B}_{k,j}$$

To determine the average parallelism for a given rule execution, the above equation is merely divided by the number of rules in the rule base:

$$P_{2-in} = \frac{\sum_{i=1}^R \sum_{j=1}^P \sum_{k=1}^R \mathcal{A}_{i,j} \mathcal{B}_{k,j}}{R}$$

The process of defining the number of Rete token collisions that occur during the execution of a given rule i is a slightly more complex task than defining the number of Rete patterns affected by the execution of a given rule. In order for a Rete token collision to occur, the following conditions must hold: For a given rule i , a token collision occurs if and only if, for any rule j in the rule base, the first two patterns in rule j are **both** affected by the patterns on the RHS of rule i . The process of detecting the number of Rete token collisions can be defined by the following pseudo-code fragment:

1. For each pair of patterns j and k in the i th column of the \mathcal{A} matrix ($\mathcal{A}_{i,j} = 1$ and $\mathcal{A}_{i,k} = 1$):
 - (a) if patterns j and k are the first two patterns in the LHS of any rule ($\mathcal{C}_{j,k} = 1$ and $\mathcal{C}_{k,j} = 1$) then add 1 to the sum of token collisions for this rule;
 - (b) else, do nothing.

Given the matrix based problem representations defined previously, the number of total number of Rete token collisions can be defined mathematically as:

$$\frac{\sum_{i=1}^R \sum_{j=1}^P \sum_{k=1}^P \mathcal{A}_{i,j} \mathcal{A}_{i,k} \mathcal{C}_{j,k} \mathcal{C}_{k,j}}{2}$$

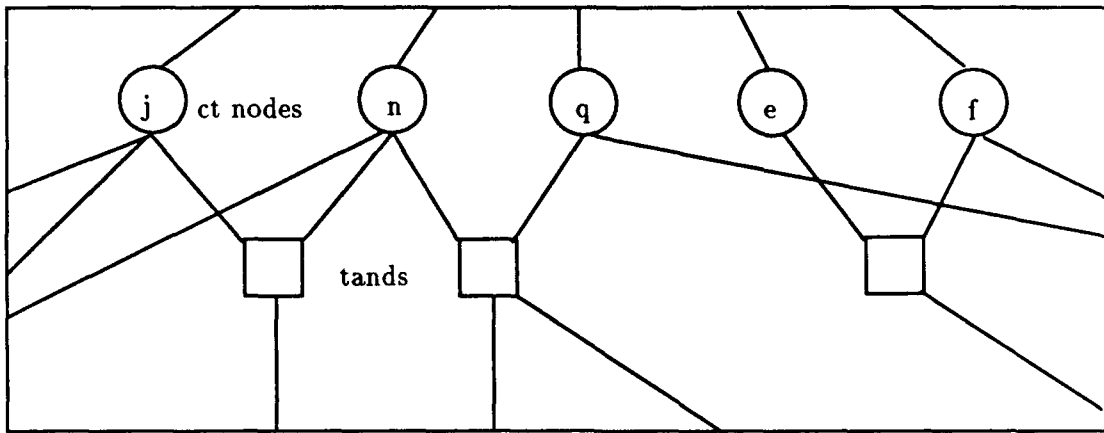


Figure 4.21. An Arbitrary Rete Network Fragment

To determine that average number of token collisions per rule execution, the above equation is merely divided by the number of rules in the rule base:

$$\frac{\sum_{i=1}^R \sum_{j=1}^P \sum_{k=1}^P A_{i,j} A_{i,k} C_{j,k} C_{k,j}}{2 \cdot R}$$

Based on definition and induction, the equation for the total number of Rete token collisions is shown to be correct:

Consider the case where rule i affects patterns j , n and q where j and n are the first two patterns in the LHS of a rule a and n and q are the first two patterns in the LHS of a rule b . Further, it is assumed that patterns j and q together do not form the first two patterns on any rule in the rule base. These initial assumptions translate to the following definitions:

$$A_{i,j} = 1 \quad A_{i,n} = 1 \quad A_{i,q} = 1$$

$$C_{n,j} = 1 \quad C_{j,n} = 1 \quad C_{n,q} = 1 \quad C_{q,n} = 1$$

This problem definition and later assumptions are accurately modeled in the Rete network fragment in Figure 4.21. Given a form of token collision equation:

$$\sum_{x=1}^R \sum_{y=1}^P \sum_{z=1}^P A_{x,y} A_{x,z} C_{y,z} C_{z,y}$$

and $i \in 1..R$ and $j, n, q \in 1..P$, the above equation yields a non-zero value only for the 4 following combinations of variable instantiations (for rule i):

$$A_{i,j} A_{i,n} C_{j,n} C_{n,j} = 1$$

$$A_{i,n} A_{i,j} C_{n,j} C_{j,n} = 1$$

$$A_{i,n} A_{i,q} C_{n,q} C_{q,n} = 1$$

$$A_{i,q} A_{i,n} C_{q,n} C_{n,q} = 1$$

Summing together the above results and dividing by two to remove the tokens that can be processed immediately produces 2 token collisions.

For any other combinations of variable instantiation values for the same rule i , the result will be 0. Therefore, the number of two-input node token collisions will not be increased. For example, no token collisions occur between pattern activations j and q for rule i because for the following variable instantiations:

$$A_{i,j} A_{i,q} C_{j,q} C_{q,j} = 0$$

$$A_{i,q} A_{i,j} C_{q,j} C_{j,q} = 0$$

none of the $C_{y,z}$ and $C_{z,y}$ values will be 1 based on the previous problem definition. Additionally, given two patterns e and f that are affected by some other rules (but not rule i according to the example assumptions), no token collisions between e and f will be created from executing rule i because for the following variable instantiations:

$$A_{i,j} A_{i,q} C_{j,q} C_{q,j} = 0$$

$$A_{i,q} A_{i,j} C_{q,j} C_{j,q} = 0$$

none of the $A_{x,y}$ and $A_{y,z}$ values will be 1 based on the problem assumptions.

Given the previous solution development, two-input node parallelism can be expressed as follows:

$$P_i = \frac{\sum_{i=1}^R \sum_{j=1}^P \sum_{k=1}^R A_{i,j} B_{k,j} - \frac{\sum_{i=1}^R \sum_{j=1}^P \sum_{k=1}^P A_{i,j} A_{i,k} C_{j,k} C_{k,j}}{2}}{R}$$

As in the previous examples for production parallelism, these equations support the assumption that each rule is executed an even number of times. To make a more accurate and realistic determination of node parallelism, some dynamic analysis of the system's execution must be available.

The upper bound parallelism described in the previous paragraphs assumes that each rule is always executed the same number of times, however we need to consider the affect of disproportionate rule executions. In Section 4.5 on production parallelism, a rule weighting factor:

$$w_i = \frac{\text{Number of times rule } i \text{ executed}}{\text{Total number of rules executed}}$$

could only be determined by actually executing the production system being analyzed. If this factor is available, the level of node parallelism in the constant-test phase can be expressed as:

$$P_c = \sum_{i=1}^R w_i \cdot \sum_{j=1}^P A_{i,j}$$

and the level of node parallelism in the two-input test phase can be expressed as:

$$P_t = \sum_{i=1}^R w_i \cdot \left[\sum_{j=1}^P \sum_{k=1}^R A_{i,j} B_{k,j} - \frac{\sum_{j=1}^P \sum_{k=1}^P A_{i,j} A_{i,k} C_{j,k} C_{k,j}}{2} \right]$$

If the w_i figures are fairly consistent for executing a number of different sets of input data then using these figures make the measure of upper bound node parallelism more accurate, but it does not allow for a more accurate measure of node parallelism.

Static analysis methods do not allow any more than an upper bound measurement of node parallelism within a given production system application. Consider again the example in Figure 4.20. Without actually executing this application and making a large number of low level measurements (requiring modification to the shell), it is not possible to determine the size of the five node activations initiated by the execution of rule 2 or whether the five Rete node activations produces any tokens that will cause activations of lower level Rete nodes (such as number 4). That is, there is no a priori way of determining how many

new token joins are created and passed to successor nodes in the network. Because the maximum number of Rete nodes decreases at each "level" in the Rete network, it is a natural supposition that the achievable parallelism is typically less than the upper-bound parallelism defined in this section. However, this supposition can only be supported by actually executing the system as static analysis cannot determine how much the lower levels of the Rete network detract from the upper bound parallelism.

4.6.4 Affect of Agenda Parallelism on Node Parallelism Parallel rule execution has the potential to significantly augment node parallelism in much the same way that it augments production parallelism. When more than one rule executes in the same production cycle, there is a significant possibility that the number of objects and the number of patterns in the Rete network affected by the most recent changes to working memory are increased. Given two sets of two-input Rete nodes, S_i and S_j , affected by the execution of rule i and rule j , the speed-up obtainable by executing these two rule during two sequential production cycles is:

$$S_{UB} = \frac{|S_i| + |S_j|}{2}$$

while the speed-up obtainable by executing the rules in parallel during a single production cycle is:

$$S_{UB} = |S_i \cup S_j|.$$

If the two disjoint rules (i and j) are executed during the same production cycle, then the potential increase in parallelism is likely to be significant; but if the two sets of patterns are merely different instantiations of the same rule, no increase in the upper-bound node parallelism will result. Therefore, although agenda parallelism may allow a large number of rule executions to take place in the same cycle, it may not have any significant effect on node parallelism. Only executing a given production system for a number of data sets determines whether agenda parallelism can significantly increase speed-ups using node parallelism.

4.7 Intra-Node Parallelism

Intra-node parallelism is another decomposition approach specific to the Rete match algorithm and possesses even finer granularity than node parallelism. Intra-node parallelism represents the lowest level of parallelism in the production by performing all tests internal to a two input Rete node in parallel (36:50). Section 2.6.2 describes, in detail, how two input Rete nodes test for consistency of condition elements by examining the cross-product of the contents in their associated right and left hand memory nodes. These cross-products can be very large for some systems thus providing a significant source of parallelism. Theoretically, combining node parallelism and intra-node parallelism allows all cross-product tests for each activated two-input Rete node to be executed in parallel (36:121). The negative aspects of intra-node parallelism include the very small grain of the tasks involved (Gupta indicates each task takes “only a few machine instructions” (36:122)) and the extra synchronization requirements beyond the synchronization requirements for node parallelism (36:50). Given the very small size of the tasks and the significant synchronization and communication requirements of intra-node parallelism, obtaining actual speed-up becomes a challenging proposition. The literature search fails to reveal any attempts to actually implement intra-node parallelism on a parallel architecture. Therefore this level of parallelism (intra-node) exists only in conceptual form.

4.7.1 Approaches to Intra-Node Parallelism Little research has been performed on applying node parallelism to increase the execution speed of production systems applications, however this investigation proposes some high-level descriptions of possible mapping approaches. In his dissertation, Gupta proposes using a system of queued cross-product tests similar to the selected approach for implementing node parallelism (Section 4.6 (36:51)). Although this approach appears to produce significant speed-ups, it appears that overhead associated with communication cost and shared resource contention is largely ignored. This controller-worker approach appears questionable because the cost of enqueueing and dequeueing cross-product tests from a shared queue is very likely to be greater than the cost of processing the cross-product test itself. As noted in Chapter 3, when the overhead cost becomes greater than the processing cost, slow-down rather than

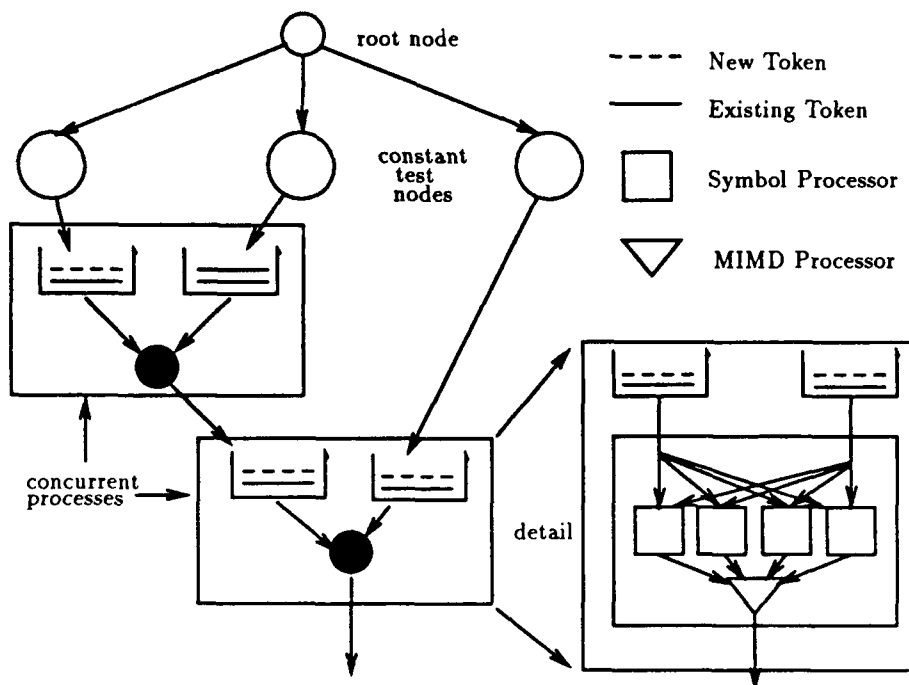


Figure 4.22. Proposed Intra-Node Parallelism Implementation Using Attached Processors

speed-up will result. As an alternative to the controller-worker approach, this investigation proposes two different approaches to intra-node parallelism: Node parallelism augmented by attached symbolic processors and direct implementation of intra-node parallelism on a SIMD class parallel architecture.

In the augmented node parallelism approach, a shared memory MIMD class machine with each processor possesses an attached symbolic processor, similar in concept to the use of attached vector processors on the Intel iPSC series computers (47). Using this approach, two input Rete node activations would be queued up and processed by a number of match processors. The main difference between this approach and the node parallelism approach is that each match processor would then use its attached symbolic processor to process the cross-product tests in parallel. Figure 4.22 illustrates how the augmented node parallelism approach processes the activations of two Rete nodes. Note that in this example the "token collision" in the leftmost two-input node means that the different tokens must be processed sequentially. Each processor uses its local symbolic processor to compare newly arrived

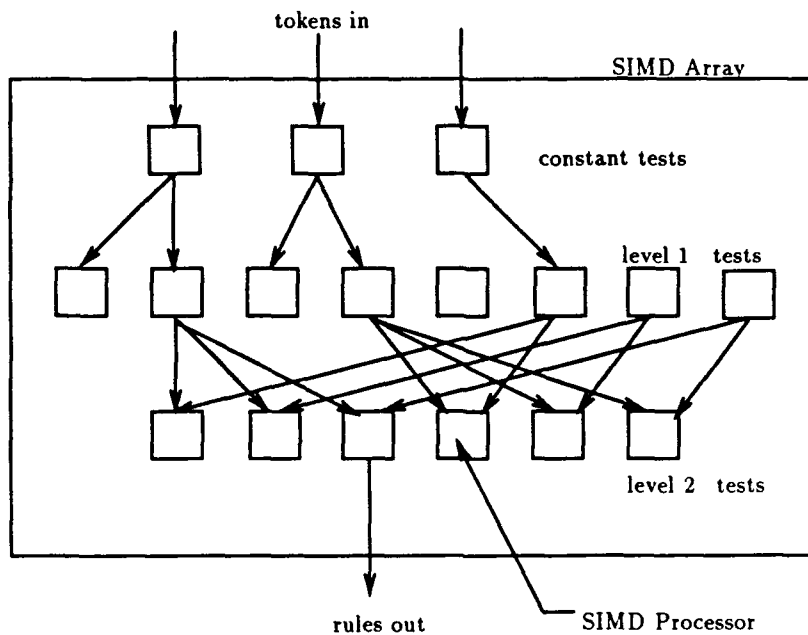


Figure 4.23. Proposed Intra-Node Parallelism Implementation Using a SIMD Architecture

tokens to all elements in the opposite side memory⁶. The one significant problem with this approach is that using an attached symbolic processor tends to reduce the granularity of the two-input match tasks significantly; as a result the queueing and dequeuing of tasks may again produce overheads that destroy most of the available speed-up.

The SIMD-based implementation proposes a very-high-level mapping of the intra-node parallelism approach to a SIMD array parallel architecture where all cross-product tests for all currently active two input Rete nodes are carried out synchronously and in parallel. Unlike the node parallelism approach that performs two-input node activations asynchronously, the SIMD-based intra-node parallelism approach processes all activations in each "level" of the Rete network sequentially before proceeding to the next "level" of activations. The assumption of synchronous operation effectively eliminates the problem of token collision, but also tends to reduce the number of two-input node activations that can be processed in parallel. Given the example in Figure 4.22, Figure 4.23 shows

⁶Existing elements have already been matched against the opposite side tokens, therefore doing so again would be a waste of processor resources

a possible mapping of the same problem to a SIMD machine. Conceptually, the problem activations could be processed in $4T_{comm} + 3T_{comp}$ where T_{comm} represents the time required to communicate the tests to be performed and T_{comp} represents the amount of time required to actually perform the tests. This simplified mapping makes many assumptions including a very significant one that the processing nodes “knows” where to send satisfied tokens. A lengthy discussion of the problems related to mapping intra-node parallelism to a SIMD type machine is beyond the scope of this effort, however it appears to be a topic worthy of future research.

4.7.2 Bounding Intra-Node Parallelism From the previous examples, bounding intra-node parallelism appears to be a relatively straightforward task, however establishing this bound assumes a significant amount of information that can only be acquired through extensive dynamic analysis. In each of the intra-node parallelism mappings discussed in the last section, the number of existing tokens and the number of new tokens for each two-input node in the Rete network is known. Study of several expert system shells reveals that this information is not available to the user without significant modifications to the source code. Additionally, the use of significant amounts of dynamic information violates the initial assumption of this investigation that parallelism would be measured using primarily static techniques. Finally, since there have been no actual implementations of intra-node parallelism, no baseline for determining the accuracy of static intra-node parallelism predictions exist. For the reasons stated above, no assessment of intra-node parallelism exists as part of this investigation.

4.8 Summary

This chapter discusses five different approaches for extracting and applying parallelism to increase the execution of production systems; it also shows that assessing the amount of parallelism in a given production system application using static methods is not an easy task. The main reason for the difficulty is that the application rule base is the only static feature around which a measure of inherent parallelism can be formed; however the production system working memory, which changes dynamically during program execution,

tends to have have a significant effect even on the measurable parallelism related to the application rule base. Of the five approaches studied, only three: application parallelism, production parallelism and node parallelism provide any basis for accurately measuring available parallelism using static analysis. The nature of the other two approaches, agenda parallelism and intra-node parallelism, defy any form of accurate measurement using static analysis techniques. The most significant negative aspect of the methods of measuring application parallelism, production parallelism and node parallelism is that they provide only an **upper bound** on the amount of speed-up that can be obtained from a specific application using parallel processing. Further research using STEPPS will tell us whether any correlation between upper-bound parallelism and realizable parallelism exists.

V. The STEPPS System Requirements and Design

5.1 Introduction

This chapter covers the salient features of the STEPPS system requirement, design and implementation as they relate to the embodiment of the parallelism measurement concepts presented in the previous chapter. As stated in Chapter 1, STEPPS provides an automated tool for analyzing the essential characteristics of production systems applications and how these characteristics relate to the successful extraction of parallelism from these applications. The actual STEPPS system described in this chapter is a self-contained system capable of accepting input in the form of production system application source code and producing an independent analysis of the characteristics and the level of inherent parallelism in that specific application. The chapter begins with a description of the STEPPS system requirements in the form of an abbreviated software requirements definition. Both object-oriented and functional design techniques are used in a complimentary manner to form the STEPPS system design presented immediately following the system requirements definition. This high-level description of the STEPPS system design provides a framework for discussing the more important details related to actually implementing the design.

5.2 STEPPS Software Requirements Document

The STEPPS system is intended only as a research tool and not as a deliverable product. Therefore, the requirements documentation in the following section provides only a concise, high-level description of the system based on the research to be carried out. Of the nine sections in a typical software requirements document, only the following four: Introduction, The System Model, Functional Requirements, and Maintenance Information (71:58-59) have significant applicability to development of the STEPPS software product. Each of these subject areas are briefly described in the following section. An additional section, Software Environment, is appended to the original list subjects areas because it contains important supplementary information that cannot be easily incorporated into any of the previously mentioned four sections. The following sub-sections may contain some

redundant information, however this is done only to the minimum point necessary to allow them to act as a stand-alone document if necessary.

5.2.1 Requirements Introduction The STEPPS system is a software research tool for investigating the level of parallelism in production system applications. The rationale and need for this tool is originally outlined in Chapter 1, Sections 4 and 5; based on this description, the purpose of the STEPPS system is to support a better understanding of how specific characteristics of different production systems applications affect the level of inherent parallelism in these systems. Using specific characteristics of production systems as the basis for parallelism measurement, STEPPS supports investigation into what "type" of applications are most amenable to parallelism using parallel computer architectures. Applications with high levels of inherent parallelism can then be used as "templates" for designers building production systems applications (most notably, expert systems) intended for execution on parallel architectures. By viewing the level of parallelism from the perspective of several different decomposition approaches, STEPPS also indicates which approach provides the most promise for maximizing speed-up on a parallel computer architecture. Experience gained from using STEPPS intends to support the future development of efficient production system shells for parallel computer architectures and applications capable of producing significant speed-ups when using these "parallel shells".

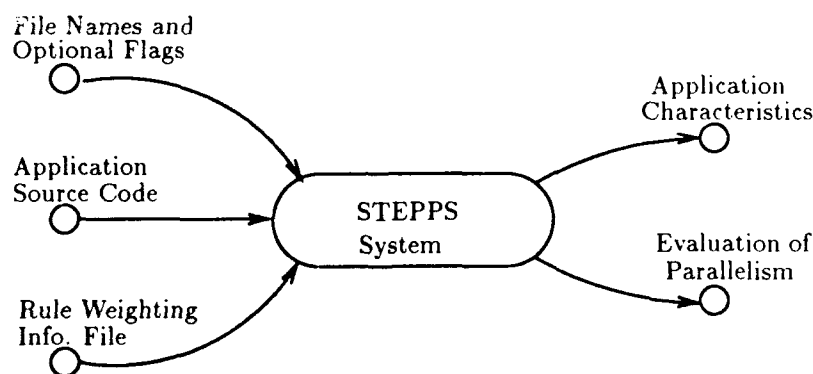


Figure 5.1. The STEPPS System Model

5.2.2 The STEPPS System Model The STEPPS system is designed as a stand-alone application, therefore its operation from the system level is fairly simple and straightforward to explain. Figure 5.1 presents the STEPPS system model as a context diagram using standard data flow diagram (DFD) symbols¹. The STEPPS user provides the names for application source code and rule weighting information files plus flags to select specific processing options. The specific format and ordering of the command-line requirements and options are presented in details in the STEPPS User's Guide in Appendix D. Production system source code is simply an ASCII format file containing all the production rules used in the specific production system application. Rule weighting information is in terms of how many times a given rule is executed in proportion to the total number of rules executed while running the given application program. The required format for each of the input files is presented briefly in section 5.2.3 and in more detail in the STEPPS User's Guide. The STEPPS system reads the command-line information and then uses this information to open and read both source code and rule weighting files (in that order). After processing the input file information, the STEPPS system produces output in the form of the production system application characteristics and the predicted upper-bound parallelism for that application. More details on the STEPPS system processing are contained in Sections 3 and 4 of this chapter.

5.2.3 STEPPS Functional Requirements The STEPPS system functional requirements are divided into the following four parts:

1. Services Provided
2. External Interfaces
3. Error Detection and Handling
4. Performance

Due to the limited time available to develop the STEPPS software product and its status as a research tool, the functional requirements reflect only the features absolutely necessary

¹For readers not familiar with DFD symbols, Chapter 12 of (71) contains a concise description of the DFD modeling process and its use in the software development process

to meet the basic research goals. By eliminating unnecessary features, the basic services provided by the system remain relatively unaffected. In order to shorten the development cycle and reduce unnecessary system complexity, the design requirements are relaxed in the areas of external interfaces, error detection and handling and in some cases, system performance. This section reveals, in general terms, the trade-off decisions between optimal design and functionality while describing the functions provided.

5.2.3.1 Services Provided by STEPPS The STEPPS system provides information that can be characterized either as *system characteristics information* or *available parallelism information*. System characteristics information is simply statistical information related the structure of the rules and how they combine to form the complete application; information that is classed as system characteristics information includes:

- Statistics concerning the number of condition elements in the right-hand-side or left-hand-side of each production rule.
- Statistics concerning the number of rules affected by the execution of a given production rule.
- Basic information such as the number of rules in the application, the average number of conditions and actions per rule and the average number of rules affected by the execution of a given rule.

Available parallelism information contains an assessment of the upper bounds on speed-up for a given production system application based on the equations in Chapter 4 of this document; information on available parallelism includes the following items:

- An assessment of the maximum speed-up possible for the given application when using the production parallelism decomposition approach.
- An assessment of the maximum speed-up possible for the given application when using the node parallelism decomposition approach.
- Information concerning the upper bounds on speed-up when implementing production parallelism using different numbers of processors.

Additional information about the contents and structure of this output data can be found in the STEPPS User's Guide.

5.2.3.2 STEPPS External Interface The external interface between STEPPS and the system user is the command-line and the data input files; this interface is not particularly user friendly, but it is sufficient to meet the needs of the system. The system user must specify the STEPPS processing options and the input file names at the time that the STEPPS system execution is invoked from the operating system command line. This interface is conceptually similar to that of a common command-line compiler. The command-line interface significantly simplifies error checking, but still provides the user with numerous processing options. The use of the command line interface also allows the output of the STEPPS system to be written directly to an output file². There are two types of data file for inputting information to STEPPS: production system application source code (psasc) files and production system application rule weighting (psarw) files. The psasc files must be in the format of one of the production system languages accepted by the STEPPS system. Currently, the two languages accepted by STEPPS are: CLIPS (versions 4.0 through 4.3) and OPS-5. Future additions may include other widely used production system languages such as ART and extensions of the currently accepted languages such as OPS-83 and CLIPS version 5.0. The psarw file contains lines of data that pair the names of rules (in ASCII string representation, on odd numbered lines) with the proportion of times the corresponding rule is actually executed (in floating point representation on even numbered lines). If no rule weighting file exists for an application, each rule is assumed to be evenly executed (i.e. the same number of times). More information on the input file format including examples, plus special cautions are provided in the STEPPS User's Guide in Appendix D.

5.2.3.3 Error Detection and Handling To simplify error detection and handling, the STEPPS system design makes several assumptions about the correctness of source code inputs and applies a somewhat "ruthless" methodology to error handling. The STEPPS system design makes the assumption that all psasc files have been successfully

²Only available when using the UNIX operating system and the ">" (redirection operator) command.

loaded into an applicable production system shell indicating that the source code input into STEPPS is syntactically correct. Making this assumption significantly reduces the STEPPS requirements to perform syntax error checking. STEPPS inputs that are not checked by another program (the command-line and psarw file data) are thoroughly error checked by the STEPPS system during program execution. If errors are detected during the execution of STEPPS, the program will either:

1. Skip over the "local" information suspected of causing the error and print out a *warning* message to the user and proceed with processing.
2. Notify the user that a fatal error has occurred and terminate any continued execution of the program.

The actual response of the STEPPS system to a perceived error depends significantly on how serious the error is. A system level error such as running out of memory or failing to find the name of an input file will result in program termination, whereas problems related to an unrecognized command-line option or an incorrect rule name in the psarw data will allow the program to continue execution. Although STEPPS error processing is significantly simplified, it is sufficient to prevent incorrect system behavior and unexpected program termination.

5.2.3.4 STEPPS System Performance Because STEPPS is not, in the strict sense, a deliverable software product its implementation does not reflect efforts to significantly reduce the time and space requirements of the system. Choices with respect to both data structures and algorithms may not always reflect optimal choices in several areas of the program; however, a review of potential system improvements using alternative data structures and algorithms indicates that the overall time order-of of the system as a whole cannot be changed from its current pseudo-polynomial status. On the other hand, it is very likely that significant increases in performance could be realized by using program profiling mechanisms to identify and reduce the size of "computationally expensive" code segments. Unfortunately, the time allotted for the STEPPS development effort does not allow the luxury of this degree of program performance optimization.

5.2.4 Software Development and Operating Environment The STEPPS system development uses the Ada programming language to implement the combined object oriented and structured analysis system design. The Ada programming language was chosen over other alternatives³ for the following reasons:

1. Direct support of object-oriented design concepts, most notably the concepts of encapsulation and information hiding(9:46).
2. Availability of generic data structures that can be instantiated with standard and user defined types (i.e. software reuse)(8:34).
3. Potential for transportability between all computer systems possessing both sufficient memory to host the source code and the capability to support a validated Ada compiler(20:6).
4. Ada is a well-defined and stable language standard whereas some of the languages under consideration are in the process of significant changes(20:2).

The development environment for STEPPS consists of the Verdex Ada Development System (VADS) version 5.5 for the Berkeley extension of the UNIX operating system, version 4.3. Only minimum modifications to the system source code should be required to host the STEPPS system on other computer and operating systems.

5.2.5 Maintenance Information The STEPPS system relies primarily on its modular design and its implementation in the Ada programming language to facilitate system maintainability and reuse. The concepts of encapsulation and information hiding are used throughout the design to minimize the impact of changes to any module on any other module. Care was taken with the overall system design so that adding capabilities to STEPPS such as adding a new production system language or adding the evaluation of a new parallel decomposition approach would require changes to very few source code module. Given the description of the STEPPS system in this chapter, the STEPPS User's Manual in Appendix D and the well documented STEPPS source code in Appendix E, a software

³The other languages considered for the development effort were: C, C++, Pascal and Scheme

designer with a good working knowledge of the Ada programming language should have no trouble in maintaining and/or modifying the STEPPS software product.

5.3 The STEPPS System Design

The STEPPS system design represents a synergistic combination of both object-oriented and structured analysis techniques, therefore both approaches are useful in the ensuing description of the high-level system design. Figure 5.2 presents the high-level DFD representation of the STEPPS system and is one level of detail "down" from the original STEPPS system context chart in Figure 5.1. This DFD reveals that the high-level structure of the STEPPS system is composed of the following sub-systems:

1. Application Parser Sub-system
2. Rule-Base Matrix Representation Sub-system
3. Parallelism Evaluation and Statistical Analysis Sub-system

The relation between these sub-systems and the system design from an object-oriented perspective is relatively easy to explain given the complete STEPPS object oriented design (OOD) in Figure 5.3. This OOD description relies on the object-oriented design notation described by Grady Booch (8:24) (9:53). The parser/processor sub-system is composed of the three left-most packages in the figure where each of these packages represent a different level of complexity in the rule base representation of the problem. The five packages in the center of the figure including the matrix package, search packages, random number package and state package make up the rule-base representation sub-system. The evaluation package is the only package that makes up the parallelism and statistical evaluation sub-system. The Command_Line package and the other packages and generics bordering the bottom of the figure are merely supporting objects. The remainder of the section describes, in detail, the design of the packages in the system OOD figure.

5.3.1 The Application Parser Sub-system The application parser subsystem is responsible for reading the production system application source code and transforming it

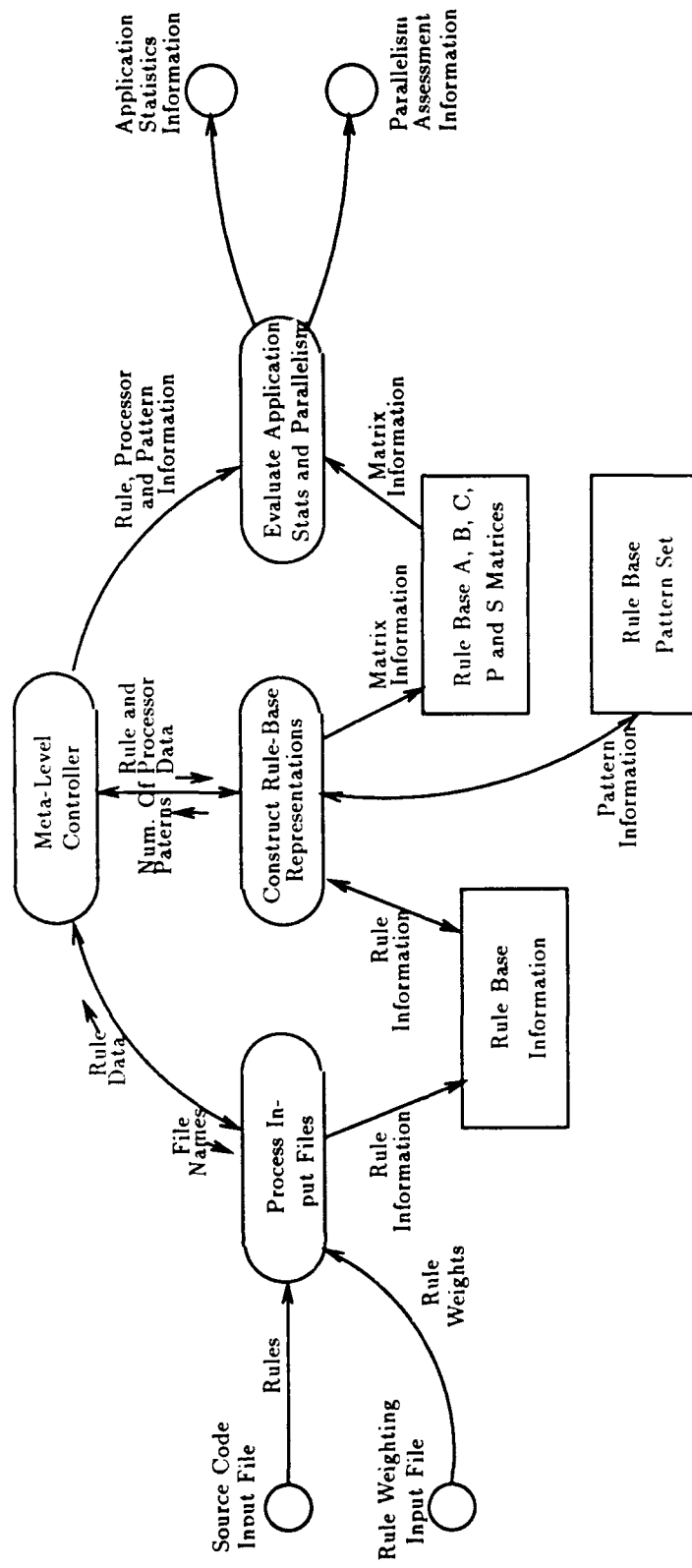


Figure 5.2. STEPPS High-Level Data Flow Diagram

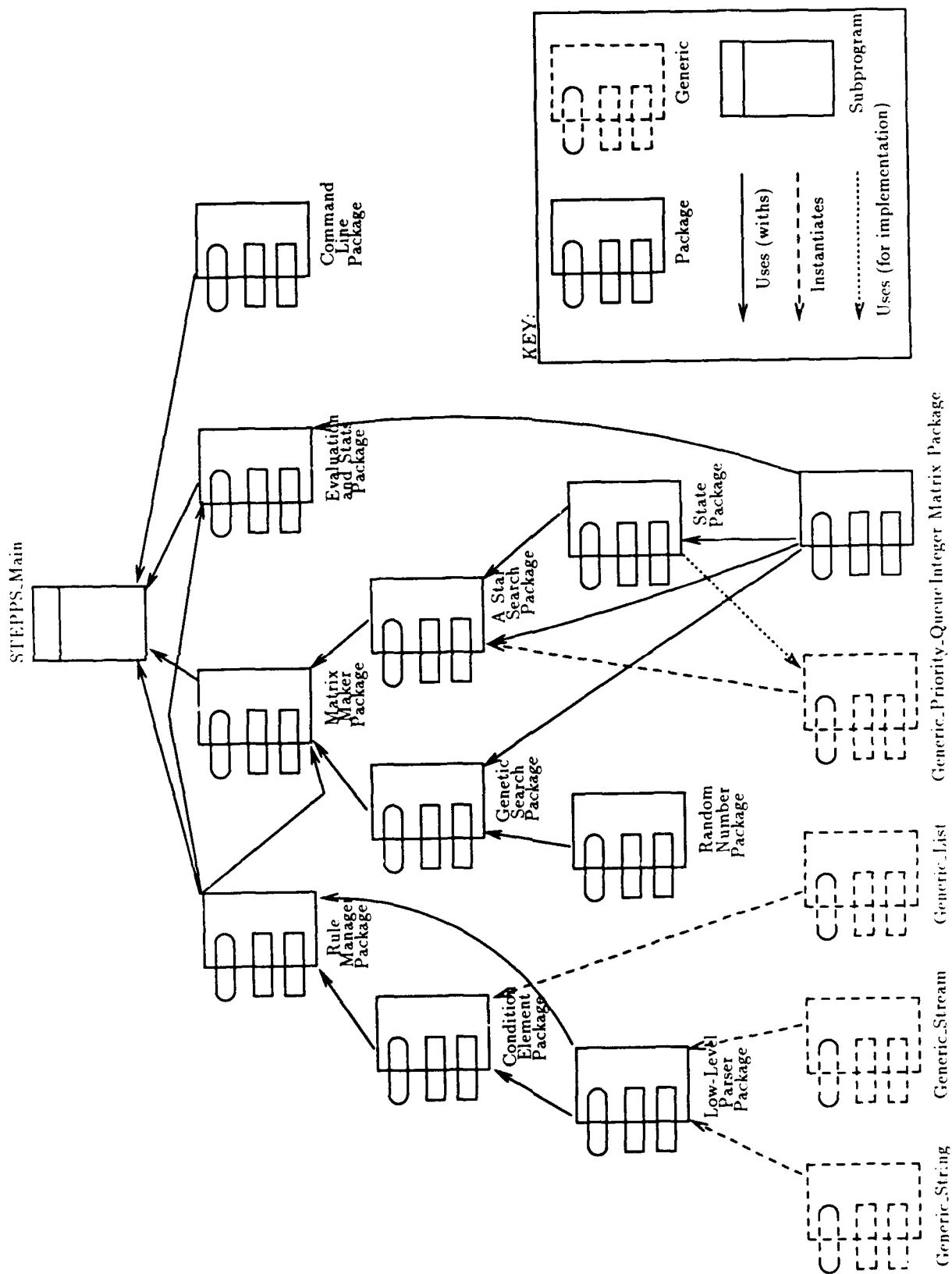


Figure 5.3. STEPPS Complete Object Oriented Design

into a representation that is useful to the STEPPS system. This transformation can be represented at a very high level of abstraction by the following three step process:

1. Get the rule from the application source code file
2. Process the rule into the required representation
3. Insert the rule into the internal STEPPS rule base

However, this three step process represents a significant simplification because the second step is conceptually very complex compared to the other two steps. The lowest level package in application parser sub-system actually “gets” the rule from the input file and the top level package in the sub-system actually “inputs” the rule into the rule-base, but actions from all three packages are required to implement the second step. Actually processing the production rules requires a number of separate steps:

1. Separate the rule into left and right hand side partitions
2. Process the left-hand-side working memory patterns into separate elements
3. Process the right-hand-side working memory actions into separate elements
4. Recombine the left and right hand side elements into a complete rule
5. Use the left-hand-side variable labels to determine which patterns are actually affected by the right-hand-side actions⁴

Details associated with processing the production rules and constructing the STEPPS rule base are covered in the implementation section.

5.3.2 Rule-Base Matrix Representation Sub-system The rule-base matrix sub-system is essentially responsible for constructing a matrix based representation of the production system application source code. The required matrix representations described in Chapter 4 are constructed as follows:

⁴Recall from Chapter 2 that variable labels are used to specify which left-hand-side pattern is affected when executing a right-hand-side *retract* or *modify* action (*assert* actions do not require labels)

1. Construct the *affect-set* (\mathcal{S}) matrix representation.
2. Construct the pattern set associated with the rule-base.
3. Construct the *affect-patterns* (\mathcal{A}) matrix representation.
4. Construct the *contains-patterns* (\mathcal{B}) matrix representation.
5. Reduce the \mathcal{B} matrix representation so that it provides an accurate representation of the application's Rete network.
6. Construct the *colliding-patterns* (\mathcal{C}) matrix representation.
7. Construct the *processor-rule* (\mathcal{P}) matrix representation for the specified number-of-processor values.

Prior to constructing \mathcal{A} , \mathcal{B} , and \mathcal{C} matrices, the *Matrix_Maker* package must construct a set of all patterns used on the left-hand-side of rules in the production system application. The pattern set information is needed to successfully construct the \mathcal{A} , \mathcal{B} , and \mathcal{C} matrices but, all other information required to construct the matrices is available directly from the STEPPS rule base. To actually form the matrices, the *Matrix_Maker* package relies heavily on functions provided by the *Rule_Manager_Package*. In order to properly construct the best possible rule to processor assignment matrix (\mathcal{P}), the matrix representation sub-system uses two different search processes. The A* search process guarantees an optimal rule to processor allocation, but it is not practical for use with production systems with more than 10 rules. For applications containing more than 10 rules, the A* search requires a prohibitively large amount of time to create the numerous \mathcal{P} matrices required by most of the applications being investigated. The genetic search package uses statistical methods in concert with simulated evolution to provide a good (but not always optimal) rule to processor allocation for larger applications. The process by which the matrices are actually constructed is examined in detail in the implementation section.

5.3.3 Parallelism Evaluation and Statistical Analysis Sub-system The parallelism evaluation and statistical analysis sub-system is the simplest of the three major sub-systems because it merely implements the parallelism evaluation equations in Chapter 4 and carries

out some averaging of the application characteristics. Specifically, the evaluation sub-system produces the following information:

1. The maximum bound of production parallelism for the application.
2. The bounds on production parallelism when using different numbers of processing elements.
3. The upper and lower bound of node parallelism for the application.
4. The average number of left-hand-side condition elements per rule.
5. The average number of right-hand-side working memory actions per rule.
6. The average number of rules affected per rule execution.

As shown in the Chapter 4 equations, evaluating the bounds on production and node parallelism requires significant rule-base matrix information plus the rule weighting information from the rule-base. Actually determining the average values for the different rule-base characteristics requires only a small amount of the rule-base matrix information, but significantly more raw rule-base information.

5.3.4 Support Packages and Main Program The remainder of the system design not covered by the specific sub-systems consists of the support packages and the STEPPS main program. The support packages consist of several generic packages and two special-purpose support packages. Generic data structures and utilities are used to facilitate software reusability in the STEPPS design and to shorten the development time required for the STEPPS system development (8:34). Each of the generic data structures aided in abstracting the conversion of the production system source code files to the STEPPS rule-base representation. The implementation section covers the use of these generics in more detail. Because most of the equations used to evaluate parallelism in production systems are written in 0-1 integer linear programming (ILP) notation, a special package was provided to provide many of the auxiliary features required of a two-dimensional matrices of natural numbers by the STEPPS system design. The main STEPPS program contains only the major sub-system calls to the packages providing the STEPPS functional requirements plus the required error-checking code for the command-line interface.

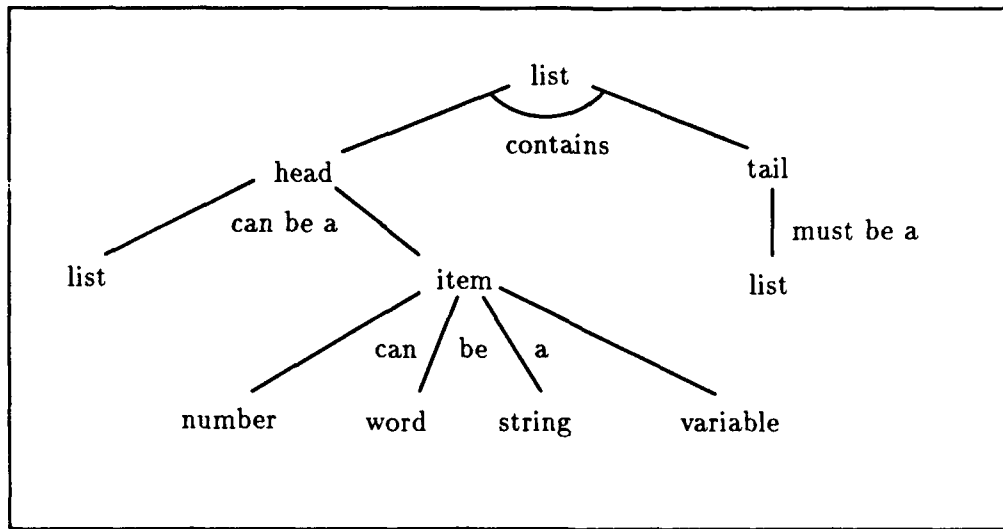


Figure 5.4. An Abstract View of the List Data Type

5.4 The STEPPS System Implementation

The STEPPS system design touches on only the high-level functionality of the system software, but this section covers the details of the system implementation that support the requirements for the system in Section 5.2 by translating the design of Section 5.3 into detailed algorithms and/or procedures and then into executable code. To avoid including too much detail, only the minimum amount of information is presented that allows the reader to “bridge the gap” between the overall system design in the previous section and the extensively commented STEPPS source code in Appendix E.

5.4.1 The Application Parser Sub-system From an abstract point of view, the application parser sub-system is only a supporting part of the STEPPS system because it automates the arduous task of extracting application specific characteristics by hand. Ironically, this sub-system is by far the most complex of the three sub-systems primarily due to the extensive and expressive syntax of the OPS-5 and CLIPS production system languages. In order to successfully process complex applications, the full rule parsing capabilities of both languages had to be implemented; this implementation was required because a subset of the language would not have allowed sufficient flexibility in the choice of applications

to test. The OPS-5 and CLIPS application code typically contains working memory facts, production rules and fact *templates*. Facts and rules were thoroughly explained in Chapter 2, but templates are used to more formally define the structure and typing of the allowable working memory facts⁵. Fact, Rule and Templates are all defined in the form of lists of items; however, from the concepts in earlier chapters, the STEPPS system is concerned only with rules. The abstraction of production rules as lists of items and lower-level lists, as illustrated in Figure 5.4, is vital to understanding how the application parser subsystem performs the task of processing rules into a representation that is compatible with system evaluation requirements.

Since production systems evolved from research in the AI field, many production system languages still use syntax associated with the most commonly used AI language known as the List Processing (LISP) language. Using LISP syntax, lists of items are represented as a left parenthesis followed by 0 or more space separated items (including other lists) and a terminating right parenthesis. Some simple examples of lists are:

```
A = (list of 4 items)      B = (item1 (first list) (second list))
```

By applying this LISP syntax, production rules can be represented as a list containing a rule keyword (such as “defrule”), the name of the rule, one or more left-hand-side patterns to match, a left-hand-side/right-hand-side differentiator (“=>” or “-->” symbols) and one or more right-hand-side actions. Following are two equivalent rules written in the CLIPS and OPS-5 syntax, note that the OPS-5 syntax allows two different means for labeling left-hand-side patterns whereas CLIPS only uses one:

<pre>(defrule example-CLIPS-rule ?x <- (monkey (holding nil)) ?y <- (stick (position on-floor)) => (retract ?y) (modify ?y (holding stick)))</pre>	<pre>(p example-OPS5-rule { <x> (monkey ^holding nil) } (stick ^position on-floor) --> (remove 2) (modify <x> ^holding stick))</pre>
--	--

⁵Templates are defined before facts and rules because they define the allowable form and syntax for these facts and rules

Using LISP type operations on production rules allows them to be more easily processed into the STEPPS representation using the process described in the following paragraphs.

Although the rules are constructed using list representation, they are stored as strings of ASCII characters in the source code input files and must be converted to the appropriate list representation. The low-level parser module is really a package for manipulating the character string representations of rules and detecting the keyword patterns associated with the syntax of the rules. Two of the more straightforward functions in the low-level parser package are the *Get_Block* and *Get_First_Field* functions. The *Get_Block* function actually gets the first top level list in a given list at a given position by matching the list parenthesis delimiters. The *Get_First_Field* function actually returns the first item encountered at a given position in a given list. The following examples, using the previously defined lists A and B, shows how these two operations are used:

```
Get_Block(B, 7) = (first list)    Get_First_Field(A, 6) = of
```

Get_Block and *Get_First_Field* are only two examples of a number of low-level string/list processing utilities. All of these low-level functions are vital to simplifying the ASCII representation of the production rules into the desired STEPPS representation stored in the rule-base.

Once a production rule is correctly extracted from the source file, it is processed in accordance with the detailed DFD in Figure 5.5. To avoid any confusion, the rule weights are actually added at a later time, but they are shown in Figure 5.5 for completeness. In the first step the rule must be first be separated into its left and right hand sides because the syntax related to each of the sides is typically very different (27) (18). Separating the rule into its left and right hand sides is very simple because of the differentiator symbol. The following actions eliminate the extraneous elements in the left-hand-side and pre-process the patterns:

1. Find and eliminate the rule definition key-word "defrule" or "p"
2. Find the name of the rule and insert it into the STEPPS rule base
3. If an internal comment block (delimited by " ") exists, eliminate it

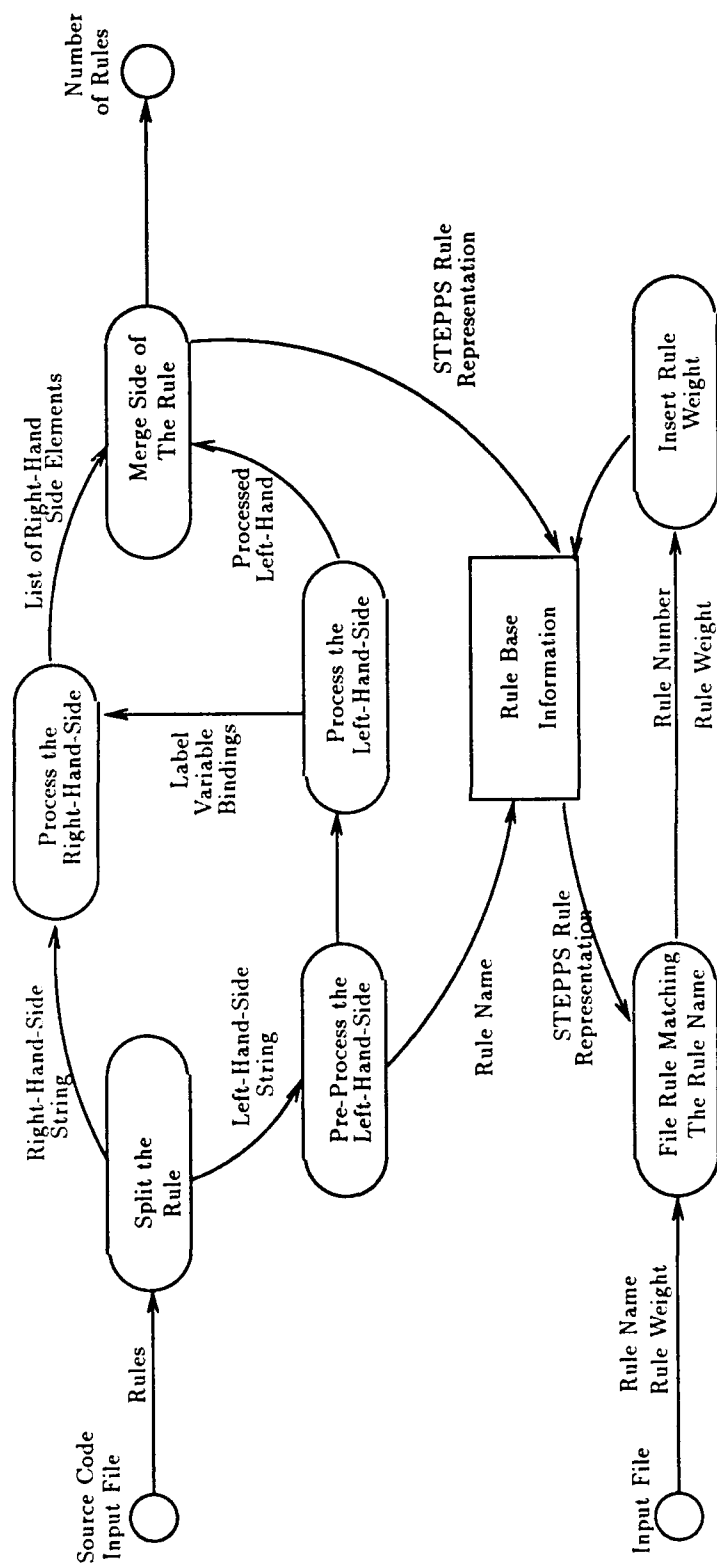


Figure 5.5. Low-Level Data Flow Diagram for the Parser Sub-System

The critical elements from each side (patterns on the left-hand-side and actions on the right-hand-side) must then be extracted before the sides can then be rejoined as a single rule and placed in the STEPPS rule-base along with the rule-name.

In addition to simply extracting the patterns and actions from the sides of the rule, a significant amount of pattern matching and replacement must be accomplished. Following is an example of a CLIPS production rule that has been processed by the STEPPS system to produce two new rule representations:

Source code rule:	The STEPPS representation of the rule:	
(defrule simple-1	(simple-1	(simple-1
(A ?x ?y)	?1 <- (A ?x ?y)	?1 <- (A ?x)
(or ?a <- (B ?x)	?a <- (B ?x)	?a <- (C ?x)
?a <- (C ?x))	?3 <- (D ?y)	?3 <- (D ?y)
(D ?y)	~(E ?x)	~(E ?x)
(not (E ?x))	=>	=>
=>	(B ?x))	(B ?x))
(retract ?a))		

There are three very important observations to make about this STEPPS rule transformation: First, the use of a two argument "or" statement in the rule reduces to two different production rule using only "and" syntax. Together, the two rules are logically equivalent to the one original rule. If the "or" statement in the above rule contained three different arguments, then three rules would have resulted instead of just the two ⁶. Second, the left-hand-side patterns with no label are given a default label corresponding to their pattern number⁷, negated patterns are not given labels because a pattern that is not actually there cannot be retracted or modified. Finally, when the left and right hand sides are again merged, the labels on the right-hand-side are replaced by the corresponding patterns from the left-hand-side. The condition element package actually performs most of the pattern processing using the list manipulation routines in the low-level parser package.

⁶CLIPS has the ability to use "or" patterns within a rule, but OPS-5 does not possess this capability

⁷The default pattern labeling is done primarily for OPS-5 compatibility

In addition to performing high-level processing on rule pattern data and adding a newly processed rule to the rule base, the rule manager package is responsible for most manipulations and queries related to the rule base. The operations performed on the established rule base by the rule manager package are:

1. Print the representation of a rule.
2. Return the string representation of the patterns in a rule.
3. Determine if executing the right-hand-side of a given rule affects the left-hand-side of another rule.
4. Determine if a given rule contains a specific pattern.
5. Determine if a given pattern is one of the first two patterns in a given rule representation.
6. Return the "weighting" of a rule with a given name (or number).
7. Return the number of patterns in the left-hand-side of any rule.
8. Return the number of actions in the right-hand-side of any rule.

All of these actions are vital to the accurate formation of the matrix representations of the application rule base and later in the actual evaluation of parallelism in the application.

5.4.2 The Rule-Base Matrix Representation Sub-system The implementation of the rule base matrix representation sub-system is fairly straightforward given the functionality provided by the rule base manager packages. For the most part, the matrix package needs only to allocate the memory space for the matrices, initialize these matrices and then use the functions provided by the rule base manager package to "fill in" the respective matrices. The definition of each of the required matrices can be found in Chapter 4, Sections 5 and 6. However, the rule base matrix representation sub-system implements three functions that do not correspond to the relatively simple process described above, these functions are:

1. Construct the set of patterns found on the left-hand-side of rules in the rule base

<code>(defrule number-1</code>	<code>(defrule number-2</code>	<code>(defrule number-3</code>
<code>(A ?x)</code>	<code>(A ?x)</code>	<code>(B ?x)</code>
<code>(B ?x)</code>	<code>(B ?x)</code>	<code>(C ?x)</code>
<code>(C ?x)</code>	<code>(D ?x)</code>	<code>(D ?x)</code>
<code>=></code>	<code>=></code>	<code>=></code>
<code>(printout "1"))</code>	<code>(printout "2"))</code>	<code>(printout "3"))</code>

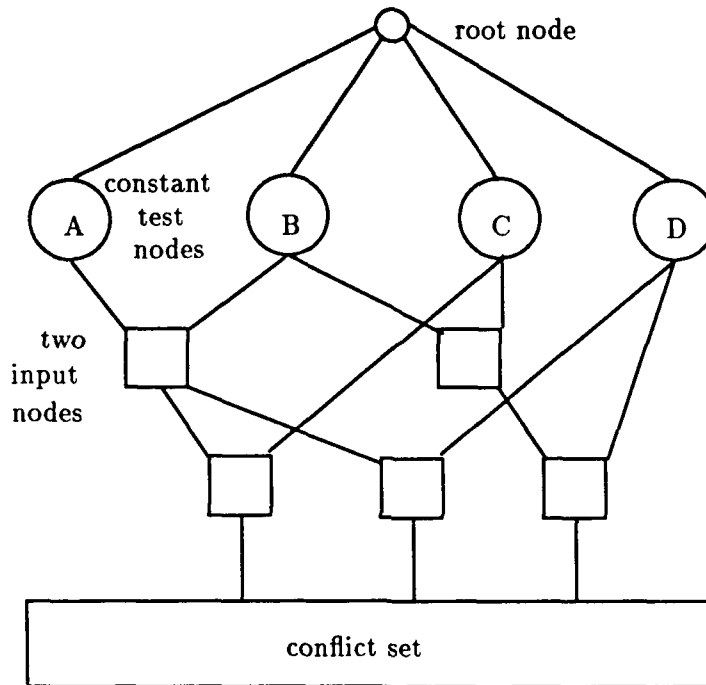


Figure 5.6. Application For Illustrating B Matrix Reduction

2. Reduce the B matrix so that it accurately represents the number of node in the Rete network
3. Search for the "best" rule to processor allocation that defines the P matrix

Each of these functions requires an algorithmic approach that is somewhat more complex than that used to actually form the S , A , B (original representation) and C matrices.

As noted in Chapter 4, Section 6.3, the original representation of the E matrix may not accurately represent the number of two-input nodes in the application's Rete network. Therefore, it must be processed to match correctly. Consider the simple example application source code and Rete network in Figure 5.6 compared to the original B matrix

$$B = \begin{vmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{vmatrix} \qquad B = \begin{vmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{vmatrix}$$

Figure 5.7. Original (left) and Reduced (right) B Matrices

in Figure 5.7. In the original B matrix, the two input "A & B" pattern appears twice whereas the Rete network sharing principle actually requires this pattern only once. Using the *reduction* algorithm, the accurate representation of the B matrix (also in Table 5.7) can be produced. A high level English description of this reduction algorithm is very complex and cannot be expressed well in concise terms, however the algorithm is very well explained in the Reduce_B_Matrix function in the Matrix_Maker_Package source code in Appendix E.

Unlike the other matrices that characterize the production systems application and can be constructed in a very simple manner, the P matrix requires a search to find the best processor to rule partitioning. Although the A^* search method is proven to be admissible⁸, it is also proven to be *NP-Complete*. Based on experience gained during the development of STEPPS, the A^* search is only appropriate for example problems of approximately 10 rules or less. It's inherent exponential time complexity renders it impractical for most "realistic" problems due to their size (typically 20 to 1000 rules). Genetic algorithms offer a polynomial time-complexity alternative to the A^* search; although search processes based on genetic algorithms (genetic search) do not guarantee admissible results(33:12). Despite not guaranteeing optimality, genetic algorithms have proven very successful in many search and optimization problems(33:212) including the rule to processor partitioning problem in this investigation. Due to the time constraints of this research project and the very small part the search algorithm plays in it, little time could be devoted to improving the initial genetic search implementation. As a direct result, the genetic search approach used to solve the partitioning problem is not likely to represent the best results that could be achieved through the use of genetic algorithms. Appendix C contains a more thorough discussion

⁸guarantees an optimal solution

of the development of both the A* and genetic algorithm search solutions to the rule to processor partitioning problem.

5.4.3 Parallelism Evaluation and Statistical Analysis Sub-system The parallelism evaluation and statistical analysis sub-system is the simplest of the three major STEPPS sub-system primarily because a majority of the system functionality has already been incorporated into the other sub-systems. Since all of the supporting matrices have been defined by the rule base matrix representation sub-system, actually implementing the parallelism evaluation equations introduced in Chapter 4, Sections 5 and 6 represents a relatively straightforward task. Additionally, the existence of statistical information in the rule base manager package also makes implementing the statistical analysis part of this sub-system straightforward. Given that the implementation of this single package sub-system relies only on implementing known equations and performing statistical averaging, no further discussion of the implementation details associated with this sub-system is necessary.

5.4.4 Support Packages A majority of the support packages STEPPS uses for its implementation are instantiations of Ada generic packages, but one of the support packages is a non-generic representation of a two-dimensional matrix. The generic packages were developed by Grady Booch as a collection of components to support software reusability (8:5); these components are available to AFIT students for use in research projects under the contract agreement with Wizard Software Inc. The generic stream package allows the application parser sub-system to get rule information from an ASCII file without having to worry about many of the details associated with file processing. The generic string and generic list packages are used by the application parser sub-system to raise the level of abstraction when constructing the STEPPS representation of the application rule base. The generic priority queue package was chosen during an earlier implementation of the A* search package(67:27). It allows for easy reusability of the code, because the A* search package imports the defined state representation (type) and instantiates a priority queue of this type to support the search process. The two-dimensional matrix package was designed and developed by the author to support the use of ILP syntax in describing the limits on parallelism in production systems. The matrix package could have been developed as a

generic package, however this would not have allowed the input/output operations on the matrix type to be incorporated into the package. Additionally, only one type of matrix was actually required for the STEPPS implementation thus implementing it as a generic package would have represented wasted effort.

5.4.5 The STEPPS Main Program Using appropriate software design principles, the purpose of the STEPPS main program is only to establish the required initial conditions, make the major sub-system calls and complete any additional support processing before allowing the program to halt execution. The STEPPS main program also intercepts all exceptions generated by lower-level procedures that are serious enough to be propagated to the top-level. When the program begins execution, the STEPPS main program calls a procedure to get all of the command-line options from the user and check them for validity. The main program then makes calls to *meta-level* procedures that control the processing of the appropriate sub-systems; the use of meta-level procedures allows the main program to remain relatively compact, hides the implementation details of the sub-systems, and eliminates the need for one of the sub-programs in each of the sub-systems from having to act as an overall controller of the other sub-programs in that sub-system.

5.5 Summary

This chapter elaborates upon the basic requirements for the STEPPS system first discussed in Chapter 1, refines and expands the design concepts discussed in Chapter 4 and provides a high-level description of the actual STEPPS implementation contained in Appendix E. The requirements for the STEPPS system reflects its status as a stand-alone research tool for investigating parallelism in production systems. The design for realizing the STEPPS system requirements uses a combined object-oriented and functional design approach; this overall design uses a multi-layer approach of system, sub-system, package and procedure components for constructing the STEPPS system. The details of the design approach, typically down to the package level are brought out in a high-level description of the implementation. In the next chapter, the STEPPS system is used to actually evaluate the level of inherent parallelism in a number of different production system applications.

VI. Investigating Parallelism Using STEPPS

6.1 Introduction

In this chapter, the previous concepts of measuring parallelism in production systems are employed. The STEPPS system is used to actually measure the level of possible parallelism in a selected set of production system applications. Using STEPPS to perform the evaluation of possible parallelism accomplishes three important objectives:

1. Evaluates the capabilities of STEPPS as a software product through testing on actual production systems applications.
2. Evaluates the accuracy and shortcomings of the metrics used by STEPPS in evaluating the level of parallelism in different applications.
3. Provides a basis for assessing the desirable characteristics of a production system application with respect to its potential for speed-up when hosted on a parallel computer architecture.

However, using STEPPS to perform parallelism evaluation requires a number of production systems applications. Given the limited amount of time available and the other requirements for this research, little time was available to locate and obtain a large number of applications to support this study.

The set of eight applications studied in this parallelism investigation come primarily from other institutions and companies involved in production systems research. The purpose of these different production systems vary considerably. A majority of the eight applications studied can be regarded as relatively simple with less than 32 production rules. However, some of the applications studied are much more complex and are capable of providing genuinely useful services. The two "monkey and bananas" applications are just pedagogical problems to demonstrate how production systems can be used to solve a simple problem. The "Digital Circuit Redesigner" and "Rubik" applications are production systems developed as class projects at different graduate schools; however, both of these systems are capable of solving limited "real world" problems. The "Tourney" production system is a simple application created to investigate specific performance problems

in production systems. The "Bogus.kb" and "Smoke1" applications are part of a parallel production system research currently underway at Merit Technology Inc. The "Robotic Air Vehicle" (RAV) is an large application intended to apply production systems knowledge representation to an air vehicle capable of autonomous operation.

The remainder of this chapter is devoted to an analysis of the different production systems applications, the STEPPS system performance while analyzing these applications and a discussion of the overall significance of the results. Each of the next seven sections in this chapter contain an analysis of a different production systems applications. This analysis is accomplished by providing a description of the system, a synopsis of the STEPPS data derived from that system and some general observations on the system design and how it affects the level of parallelism in the application. The STEPPS evaluation section contains an discussion of the capabilities and limitations of the STEPPS system discovered during the parallelism investigation. Finally, the overall results section combines the finding of the seven applications analysis sections in a coherent manner.

6.2 *The Monkey and Bananas Problem*

The monkey and bananas (MAB) problem is both a simple and popular exercise in knowledge representation and planning control in AI systems. Figure 6.1 aids in describing the elements of the MAB problem and approaches for solving it. Basically, the problem begins by placing the following items in a closed room:

- A heavy (weight) couch placed against one wall of the room
- A light (weight) chair placed against another wall of the room
- A light (weight) step-ladder placed against yet another wall
- A short (length) stick placed at an arbitrary position on the floor
- One bunch of bananas suspended from a hook in the center of the room's ceiling
- One very hungry monkey who cannot lift heavy objects

The goal of the problem, of course, is for the monkey to use the various items at his/her disposal in order to get the bananas down from the hook and consume them. The above

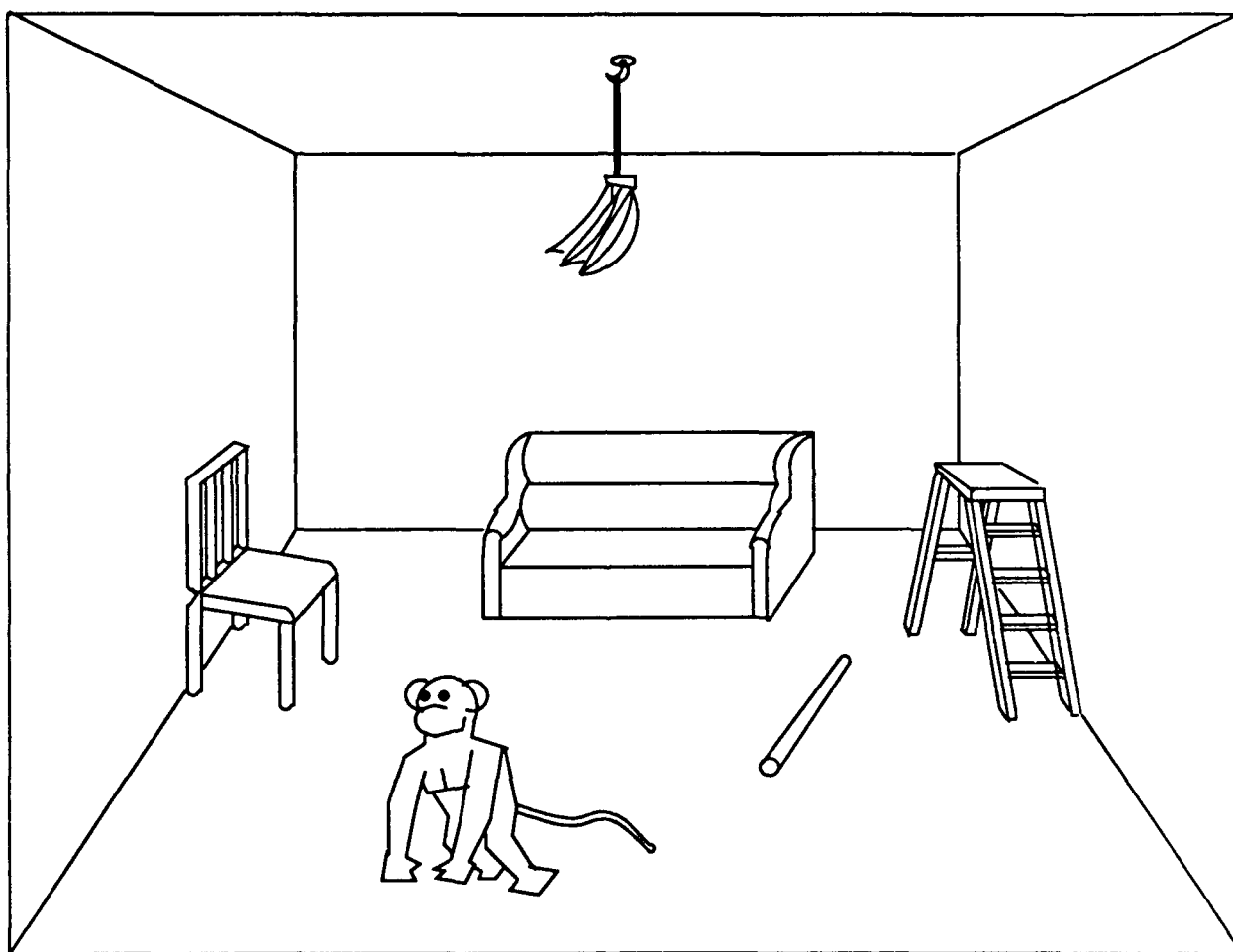


Figure 6.1. A Visual Representation of the MAB Problem

description is very general as it has been adapted for use by a number of researchers. This section analyzes two different implementations of the MAB problem. The first is written in OPS-5 and is based significantly on the MAB problem implementation defined by Brownston (14:383-408). The second MAB problem implementation is a significant extension of Brownston's MAB problem written in CLIPS by researchers at JSC (65).

6.2.1 The OPS-5 MAB Implementation The original monkey and bananas implementation is an example of a *tightly coupled* rule base. By *tightly coupled*, it is meant that the execution of any given rule affects a majority of other rules in the rule base. The 20 rules contained in this implementation have an average of 2.4 LHS conditions per rule and 1.9 actions per rule. Despite the small number of actions per rule, that average number of rule affected by the execution of any given rule is 19. The "start" rule is executed only once and is not affected by the execution of any other rule; therefore, for all practical purposes, each rule execution affects the LHS of every other rule. The tightly coupled nature of the application produces large estimates (with respect to the rule base size) for both production parallelism and node parallelism. The CMU node parallelism implementation for the Encore Multimax multi-processor produces a maximum speed-up of 1.3 achieved with 5 processors. This figure is much less than upper-bound speed-up of 17.2 predicted by STEPPS. However, the simplicity of the rule base itself makes most tasks very small grained and, therefore, tends to limit achievable parallelism.

6.2.2 The CLIPS MAB Implementation The CLIPS version of the monkey and bananas problem reflects many of the characteristics of the original OPS-5 problem description. This particular problem implementation contains 32 rules with an average of 3.2 conditions and 3.1 actions per rule. As with the original implementation, the rule base is tightly-coupled. The average number of rules affected by the execution of any given rule is 30.94. Again, the upper bound parallelism predictions are high. The estimate for the upper-bound production parallelism is 26, and the estimate for the upper-bound node parallelism is approximately 35. Researchers at JSC have achieved speed-ups of 1.5 using 4 processors and a production parallelism implementation of CLIPS on a FLEX multi-processor (65:386). STEPPS predicts an upper bound speed-up production parallelism

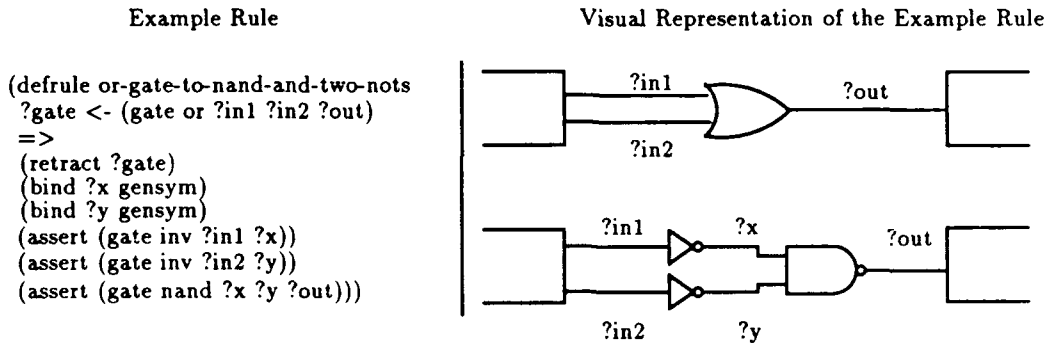


Figure 6.2. Example Digital Circuit Redesign Transformation Rule

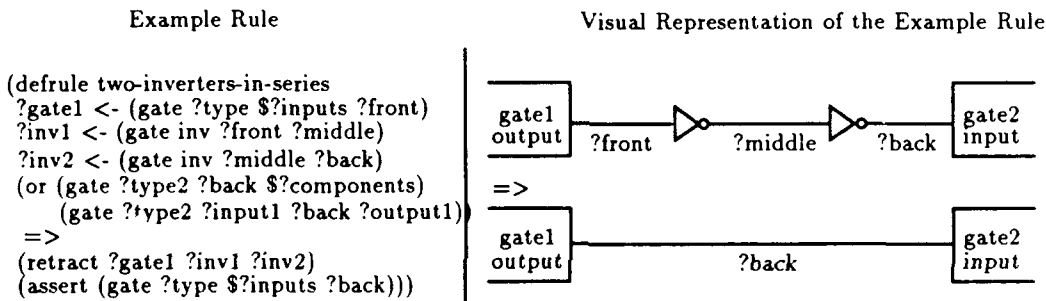


Figure 6.3. Example Digital Circuit Redesign Simplification Rule

match phase speed-up of 3.97 when using 4 processors. Assuming that Rete state update phase requires 90% of the total processing time, this translates to an overall upper-bound speed-up of approximately 3.68. The speed-up actually achieved by JSC researchers is still much less than the 3.68 upper bound value, but it is greater than the $3.68/5 = 0.736$ value predicted by Gupta based on his observations of production parallelism execution models (36:49).

6.3 The Digital Circuit Redesign System

The Digital Circuit Redesigner (DCR) application performs local redesign on digital circuits for VLSI design compatibility and circuit simplification. The DCR system began as a classroom exercise, but has been expanded significantly in an effort to aid VLSI circuit designers. Using the pattern matching facilities of CLIPS, DCR transforms the

original circuits into circuits containing only NAND and inverter gates which form the basis for most VLSI designs; an example circuit transformation rule is shown in Figure 6.2. DCR also performs simple circuit reductions on the transformed circuits using local optimization techniques. An example circuit optimization involves the elimination of two inverter gates in series (and no other connections between them); Figure 6.3 illustrates the CLIPS and visual representation of this particular circuit optimization. Additionally, the DCR system provides simulation rules to double-check that all circuit transformations have been done correctly using a "before = after" test. The current DCR implementation is relatively simple, but future additions are expected to make it more useful to VLSI designers. Among the proposed changes are:

- Addition of timing constraints to the local circuit transformation rules.
- Addition of a "heuristic" search process to provide better "global" circuit optimization.
- Pre and Post processors to allow compatibility with Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL).

The DCR system possesses characteristics that are somewhat different from either of the monkey and bananas problems discussed previously; however, some similarities do exist. The DCR system consists of 45 rules with an average of 3.76 condition per rule and 3.91 actions per rule. Gupta notes that a larger number of actions immediately implies a greater possibility for parallelism, because many changes to working memory can be processed in parallel (36:24). Therefore, the DCR system's potential for parallelism appears to be better than either of the monkey and bananas problems. However, the average number of rules affected by the execution of a given DCR system rule is only 18.5 or 41% of the rule base compared to the 90%+ of the monkey and bananas problems. In this respect, the DCR system appears to measure up poorly compared with the previous example applications. For the DCR system, STEPPS predicts an upper-bound production parallelism of 12.38 and an upper-bound node parallelism of 21.5. The graph in Figure 6.4 illustrates that the upper bound on production parallelism increases steadily for between 1 and 12 processors, but levels off rapidly for more than 12 processors. Increasing the

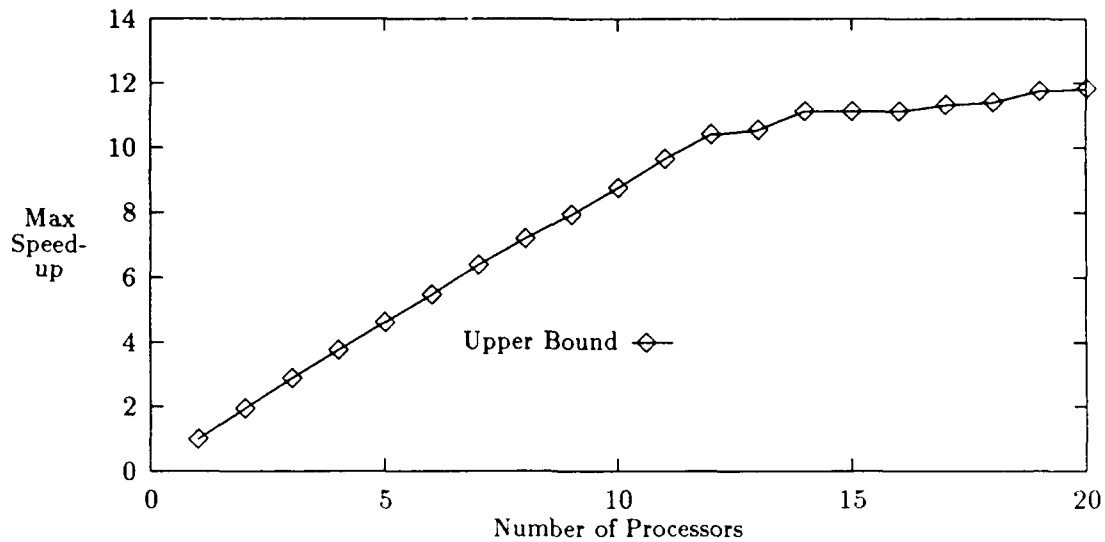


Figure 6.4. Production Parallelism Estimates for the DCR System

number of processors does not yield significantly higher levels of production parallelism due to the effects of rule partitioning and dynamic system information introduced in the form of rule weighting assignments.

6.4 The Bogus SDI Example Scenario

The Bogus.kb is a production system developed by Merit Technology to roughly simulate the information rich domain of SDI contact discrimination. The following description is taken directly from (7:4-5):

“The Bogus knowledge base contains three types of facts:

- tester – which has two attributes, *type* and *slot*. There were four initial assertions of this fact that vary in the value of *type* from 0 to 3. The *slot* attribute of each fact is initially set to 0 and incremented by 10 during each production cycle.
- cycle counter – has a single attribute, *value* that is initialized to 0 and incremented by 1 during each production cycle. Each other rule requires the existence of this fact in order to be satisfied.

- max cycle – has a single attribute, *count* that is initialized to some arbitrary preset value that allows the retraction of the cycle counter fact when the *value* of the cycle counter fact matches the *count* of this fact.

The Bogus rule base contains the following types and numbers of rules:

- 1 - cycle counter rule increments and tests the value of the cycle counter fact during all but the last production cycle. If “k” production cycles were executed, this rule would fire “k - 1” times.
- 1 - stopper rule executed during the last production cycle in place of the cycle counter rule and deletes the cycle counter fact thus making all other rules ineligible to fire.
- 8 - body rules, labeled rule_i_j where i varies between 0 and 3 and j varies between i+1 and 4 (except for rule_1_2 described below). Each of these rules tests the *value* of the cycle counter fact; Two different tester fact’s *type* attribute, corresponding to the “i” and “j” labels of the rule and the *slot* values of any two tester rules. Each body rule satisfied creates two new tester facts of *type* identical to the “i” and “j” values of the rules. The interaction of these rules and the facts in working memory generates a rapid explosion of facts that subsequently cause many more rule instantiations to be satisfied during the subsequent cycle. When firing rules in parallel, each body rule is executed once the first cycle and 10 times the second cycle with the number of rules fired growing at one order of magnitude during each production cycle.
- 1 - typo body rule, labeled rule_1_2 contains one additional condition when compared to the above body rules. This condition allows it to be executed only once during each production cycle instead of increasing the number of times it is fired during each subsequent production cycle.
- 100 - body filler rules labeled rule_x_j where “x” is a lower case alpha character and “j” varies between 0 and 4. These rules are almost identical to the body rules except that they test cycle counter fact *value* attributes and tester fact *slot* attribute values that are never satisfied. These rules are intended to simulate the processing load of a much larger system.”

In a typical Bogus.kb scenario, six initial facts are asserted. These initial facts result in 59 rule firings in two production cycles when using the METE shell to fire the satisfied rule instantiations in parallel (7:5). Minor changes to the internal structure of some rules in the rule base are required when reimplementing Bogus.kb in the CLIPS production system language. These required changes were made only to allow the same results to be obtained when the rules were fired during sequential production cycles as opposed to firing them in parallel during the same cycle.

The Bogus.kb application may initially appear to possess significant levels of parallelism; however, this appearance is somewhat of an illusion. Although Bogus.kb contains 108 rules, the CLIPS representation only “sees” 9 unique rules. The patterns on the LHS of the 100 body-filler rules are essentially the same, thus CLIPS uses the same set of Rete nodes to represent these rules. Although the upper-bound production parallelism is 108, this value is misleading because only the “cycle-counter”, “stopper” and “body” rules are ever executed. The one extra “body filler rule” which represents the same patterns as all of the other “body filler rules” is never executed. Given the a priori information that the “body filler rules” are never executed, the upper bound production parallelism is now 11.3 as only the 12 “core” rules (the ones executed) plus one “body filler rule” are included in the evaluation. The STEPPS system uses the same approach to Rete pattern evaluation as CLIPS; therefore, the upper-bound node parallelism value of 15 can be considered accurate with respect to the CLIPS implementation. Merit Technology has achieved speed-ups of 29 for Bogus.kb using the “Meritool” shell hosted on a BBN Butterfly multi-processor system. This result does not appear to be supported by the STEPPS analysis. However, Merit Technology’s METE algorithm does not conform to the node parallelism model derived in Chapter 4, Section 6.2. Additionally, the METE algorithm executes multiple rules in parallel which can significantly increase available parallelism¹. Given the incompatibility of the STEPPS model with the METE algorithm and the perturbations introduced by parallel rule execution, the actual results are explainable.

6.5 The Tourney Scheduling System

Tourney is a program for assigning match schedules during a (generic type) tournament developed by Bill Barabash of the Digital Equipment Company (40:111). Judging from the schedule produced by the program, the tournament scheduled is a *round-robin* type where every team plays every other team and the team with the best record is the overall winner. In this type of tournament, if multiple teams finish with equally good records, a subsequent single-elimination round is played to determine an overall winner. For instance, given the Tourney-produced schedule in Figure 6.5 in the first round, team

¹See Chapter 4, Sections 5.5 and 6.4 for details.

Tournament Schedule																			
					N S E W										N S E W				
					= = = =										= = = =				
#1:	Group A:	13	12	4	3	# 6:	Group A:	13	4	12	3	#11:	Group A:	13	3	4	12		
	Group B:	14	11	9	7		Group B:	14	9	11	7		Group B:	14	7	9	11		
	Group C:	15	10	5	2		Group C:	15	5	10	2		Group C:	15	2	5	10		
	Group D:	16	8	6	1		Group D:	16	6	8	1		Group D:	16	1	6	8		
#2:	Group A:	12	14	5	1	# 7:	Group A:	12	5	14	1	#12:	Group A:	12	1	5	14		
	Group B:	4	15	8	7		Group B:	4	8	15	7		Group B:	4	7	8	15		
	Group C:	3	16	9	2		Group C:	3	9	16	2		Group C:	3	2	9	16		
	Group D:	13	11	10	6		Group D:	13	10	11	6		Group D:	13	6	10	11		
#3:	Group A:	14	16	10	4	# 8:	Group A:	14	10	16	4	#13:	Group A:	14	4	10	16		
	Group B:	5	11	8	3		Group B:	5	8	11	3		Group B:	5	3	8	11		
	Group C:	1	15	13	9		Group C:	1	13	15	9		Group C:	1	9	13	15		
	Group D:	12	7	6	2		Group D:	12	6	7	2		Group D:	12	2	6	7		
#4:	Group A:	16	15	12	11	# 9:	Group A:	16	12	15	11	#14:	Group A:	16	11	12	15		
	Group B:	10	7	3	1		Group B:	10	3	7	1		Group B:	10	1	3	7		
	Group C:	4	9	6	5		Group C:	4	6	9	5		Group C:	4	5	6	9		
	Group D:	14	13	8	2		Group D:	14	8	13	2		Group D:	14	2	8	13		
#5:	Group A:	15	14	6	3	#10:	Group A:	15	6	14	3	#15:	Group A:	15	3	6	14		
	Group B:	12	10	9	8		Group B:	12	9	10	8		Group B:	12	8	9	10		
	Group C:	11	4	2	1		Group C:	11	2	4	1		Group C:	11	1	2	4		
	Group D:	16	13	7	5		Group D:	16	7	13	5		Group D:	16	5	7	13		
End of Scheduling																			

Figure 6.5. A Example Schedule Produced by Tourney

13 plays team 12, team 14 plays teams 11, team 15 plays team 10, and team 16 plays team 8. At the end of 15 rounds, each of the 16 teams in the tournament has played each other team exactly once. Pattern matching associated with the Tourney scheduling process is very extensive and causes the program to run very slowly even though the application is very small.

The Tourney system is a fairly small sized application (17 rules). But in terms of other characteristics, it is similar, in many respects, to the DCR system discussed earlier in this chapter. Rules in the Tourney system have an average of 5.06 conditions per LHS and 4.41 actions per RHS. Executing a Tourney rule affects an average of 10.53 other rules or approximately 61% of the rule base. Based on analysis of the DCR system, it is possible to hypothesize that the upper bound on production parallelism will be significantly less than the average affect set (rules affected by a rule execution) size. This assertion again proves true as the upper bound production parallelism figure produced by STEPPS is 4.27 for 6 processors. As the graph in Figure 6.6 illustrates, production parallelism cannot be increased by (and may actually be harmed by) adding additional processors. The STEPPS predicted upper bound on node parallelism is approximately 13.5; however, CMU's node parallelism implementation fails to achieve a speed-up of even 3. There are several important reasons why the actual implementation performance fails to approach the upper-bound node parallelism prediction.

The Tourney system is actually a combination of very complex and very simple production rules which yield the *average* characteristics described in the previous paragraph. CMU researchers note that combinations of simple rules executed in sequence (short-chain effect) and complex rules, with large amounts of data, (cross-product effect) can combine to severely limit the amount of available parallelism, as in the case of Tourney (40:117). Although this information helps explain why the STEPPS evaluation varies so significantly from an actual implementation result, it does not provide a firm basis for understanding when this problem occurs or for predicting the extent of its adverse affect on upper-bound parallelism. According to CMU researchers, problems related to short-chain effect can only be avoided through careful application design (74). The effect of short-cycles may be reduced by identifying them and performing processing for those cycles on a single pro-

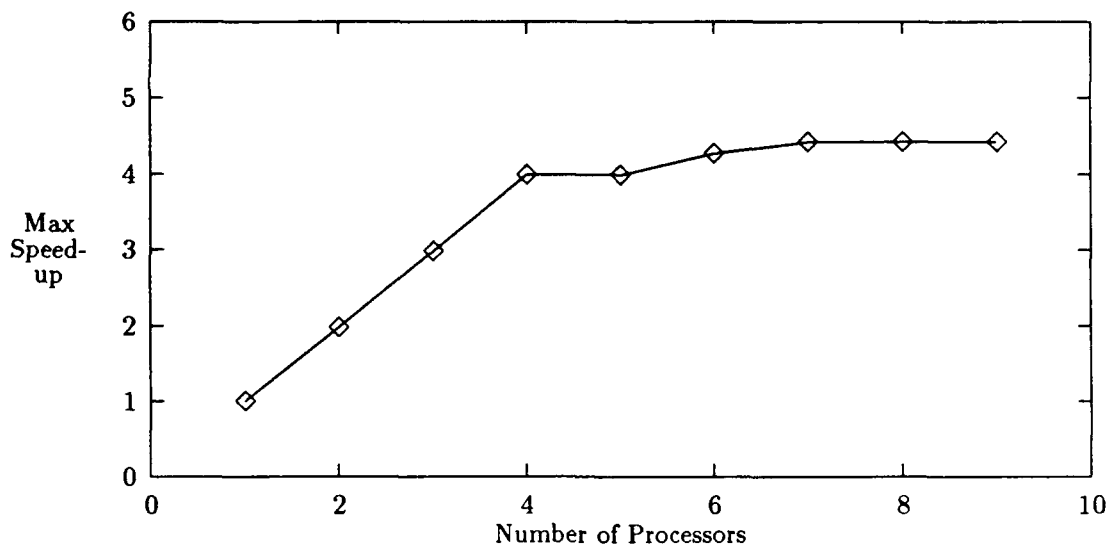


Figure 6.6. Production Parallelism Estimates for the Tourney System

cessor as opposed to multiple processors (75:16). However, this approach only lessens the short-cycles effect, it does not solve it. Although the cross-product effect can sometimes be avoided through careful design, the effect of cross-products can also be lessened by splitting single Rete TANDs into two or more TANDS (see Figure 6.7) (39:690). Using this approach, parallel simulations of the Tourney system have shown node parallelism speed-ups of almost 8 times, which is much closer to node parallelism upper bound prediction of 13.5 (75:20). However, Tambe notes that the success of this method sometimes requires significant Rete network data-flow analysis (74). Despite the fact that short-chains and cross-products can reduce speed-ups significantly, static analysis techniques such as those used by STEPPS cannot accurately detect them and quantify their effect on parallelism.

6.6 *The Rubik's Cube Problem Solver*

Rubik is a production system application developed by James Allen of CMU to solve the popular "Rubik's Cube" problem (40:111), a popular three dimensional puzzle. The program begins by allowing the user to specify the original arrangement of the cube before solving begins; this process of reorienting the faces of the cube is also known as

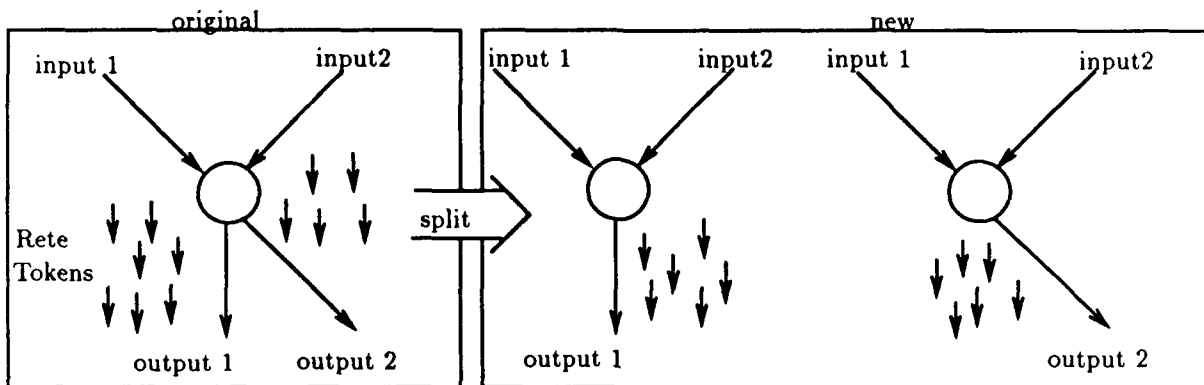


Figure 6.7. An Example of Splitting Two Input Rete Nodes (75:16)

“scrambling”. Once the cube is scrambled, the Rubik program uses some simple, but very powerful heuristics for solving the problem. The program listing shows the significant amount of pattern matching required for this problem and also describes some of the powerful “rule of thumb” heuristics used by human “experts” in solving this problem. The magnitude and complexity of the production system code indicates that solving this problem using admissible techniques, such as A* would be very difficult.

Rubik can be classified as a tightly coupled system, but it is a very complex system and possesses characteristics that are significantly different than the other applications evaluated in this chapter. The Rubik system’s complexity can be attested to by the average of 9.37 LHS conditions and the average of 6.96 RHS actions of the 69 rules in the rule base. Both of these average characteristics are significantly higher than for the applications evaluated thus far. Each Rubik rule execution affects an average of 50.91 or approximately 72% of the rule base. This means Rubik’s actual level of rule coupling falls somewhere between the 90% rule coupling of the monkey and bananas example problem and the 40% to 60% rule coupling of the DCR and Tourney applications. Based on the complexity and rule coupling of the Tourney application, it might be expected that the upper-bound parallelism predictions will be relatively large with respect to the application size, and they are. The upper bound on production parallelism for Rubik is 46.12 and the predicted upper bound on node parallelism is approximately 135 to 140! This level

of node parallelism may appear unrealistically large; however, it does indicate significant possibility for relatively large speed-ups.

The primary reason for Rubik's high level of node parallelism is that it contains over 400 different Rete network patterns many of which are affected during each production cycle. In a manner similar to the monkey "object" in the monkey and bananas exercise, the physical cube is the object of interest in the Rubik system. Unlike the state of the monkey "object" which only possesses a few attributes (such as position, hungry-for, holding, etc.), accurately capturing the state of the cube requires a large set of very complex facts. The rules manipulating the cube, which constitute a majority of Rubik's rule base, reflect the complexity of the cube's representation. The extensive pattern matching required to satisfy rules and carry out the appropriate actions create a large number of tasks that can be executed in parallel (40:116). This fact is reflected in the almost linear speed-up achieved for the Rubik system using CMU node parallelism implementation on an Encore Multi-Max multi-processor system (40:118). Of the system examined thus far, Rubik appears to be the best suited for node parallelism and possibly for production parallelism as well.

6.7 The Smoke1 Pollution Control Expert

Smoke1 is an expert system which aids the State of South Carolina environmental managers in controlling air pollution caused by the burning of vegetative debris. Specifically, the system is used to review requests for vegetative burning permits within the state of South Carolina. Permits for burning, and the time burning is permitted are based on a number of factors such as:

- The material(s) to be incinerated
- Under what conditions the material will be burned
- The location of the burn with respect to smoke sensitive areas
- Weather factors at the burn site
- The amount of air pollution caused by permits already granted

The Smoke1 application was originally developed by Jim Saveland of the University of South Carolina; this original implementation is a backward chaining expert system using the *IBIS* shell. In the backward chaining mode, Smoke1 is an interactive system that asks the user relevant questions about the requested burn permit and then provides an recommendation of whether or not this burn should be permitted. Researchers at Merit Technologies made changes to the original program that allows Smoke1 to run in a forward chaining mode under the "Meritool" shell. In the forward chaining mode, the user inputs a number of permit requests, each containing all information about the requested burn. Smoke1 then processes each request in order of receipt and provides a accept/reject answer for each request along with the acceptable burning times if the request is approved. The Smoke1 application was recoded in the CLIPS production system language expressly for this research work.

Smoke1 is yet another example of a tightly coupled rule base as the execution of a given production rule affects nearly every other rule in the rule base. Each of the 87 Smoke1 rules contain only 1 LHS condition and an average of only 1.22 RHS actions. Despite this, executing any one of the Smoke1 rules affects all 87 rules in the rule base. STEPPS evaluates the upper-bound production parallelism at 86.9 and the upper-bound node parallelism at 86.0. The reason for these high levels of parallelism is that Smoke1 has one class of fact, called "data", which contains a large number of different fields to be matched. This particular fact class is what makes up the LHS of each rule. Each Smoke1 rule contains a different Rete network pattern which results in the equally high level of node parallelism. Although Smoke1 may appear to contain large amount of parallelism, the size of the match tasks are likely to be very small because each rule contains only one pattern. Smoke1's small task sizes are likely to require combining a large number of rules on each processor when using production parallelism in order to amortize interprocessor communication. Although the level of node parallelism appears to be significant, the time required to enqueue and dequeue Smoke1's small sized tasks Smoke1 may actually be greater than the time required to process the activation. This situation could potentially produce slow-down instead of speed-up.

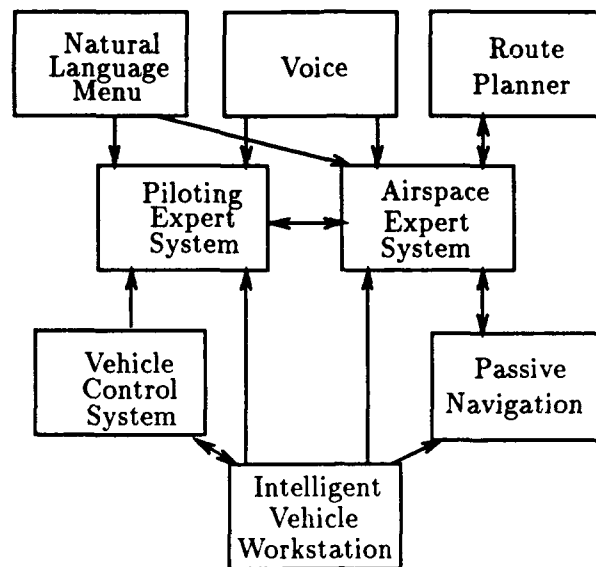


Figure 6.8. The Robotic Air Vehicle (RAV) System Architecture (57:1327)

6.8 The Robotic Air Vehicle System

The Robotic Air Vehicle (RAV) represents an attempt by the Defense Advanced Research Projects Agency (DARPA) and the Wright Research and Development Center (WRDC) to create an unmanned flight vehicle capable of autonomous operation (41:A-1). The initial RAV feasibility study was awarded to Texas Instruments in 1985 and this original research has continued to the present (25). As Figure 6.8 illustrates, the current RAV architecture consists of six top level modules including two expert systems. The Piloting Expert System (PES) is the top level controlling agent in the RAV architecture and provides high-level commands to the other modules (41:A-1). Other systems, such as the Vehicle Control System (VCS) translate the high-level commands from the PES and carry out algorithmic based procedures based on the input command(s) (41:A-2). For example when the following high-level PES rule executes:

(rule rotate-and-leave-ground
 (flying speed reached)
 (systems status OK)

=>

(assert (rotate-to 15 degrees nose up)))

the VCS receives the fact assertion and increases the angle of attack of the RAV's elevator slowly (2-3 degrees per second) until the attitude indicator reads 15 degrees nose-up. In addition to basic vehicle control skills (including aerobatics), the PES is responsible for passive terrain following, terrain avoidance, obstacle avoidance and autonomous navigation (41:A-1). The current version of the RAV is written in ART, but was transliterated into CLIPS by William A. Harding in conjunction with his thesis research.

The RAV's PES is the largest application investigated. However its level of inherent parallelism appears to be less than that for the smallest application (Tourney) studied. In contrast to the tightly and moderately coupled systems rule bases of the other application, the PES rule base is very loosely coupled. Executing a single PES rule will only affect an average of 9.527 other rules or approximately 3.4% of the rule base. The PES's 277 rules contain an average of 2.87 conditions and 1.91 actions which may seem surprisingly simple given the complex task of flying an aircraft. The very loose coupling of the PES rule base is reflected in the following upper-bound speed-up evaluations: The production parallelism upper-bound value is 9.53 and the node parallelism upper-bound value is 9.62. The upper bound for the production parallelism approach using between 1 and 5 processors is shown in Figure 6.9. This Figure shows that the **upper bound** for 5 processors or less is very low, thus indicating that the actual attainable parallelism using a production parallelism shell would be even lower.

The upper-bound parallelism results for the RAV rule base help to explain, in part, the past failures in effectively applying parallel architectures to this problem. Past research in applying production parallelism approaches to PES processing have produced speed-ups as high as 5.1 for a subset of the total rule base (42:24) and negative speed-ups² when using the entire rule base (42:26). Applying STEPPS to various portions of the PES reveals that functionally grouped subsets of the rule base possess higher rule coupling than the rule

²Speed-up values were less than 1.

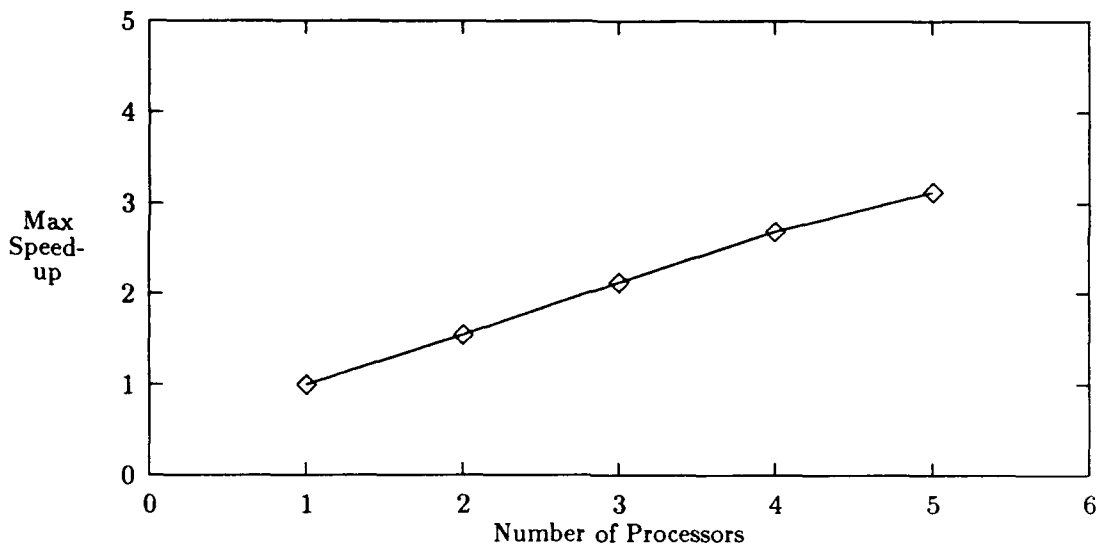


Figure 6.9. Production Parallelism Estimates for the RAV System

base as a whole. There is the additional consideration that the best speed-ups for the RAV application do not utilize the optimal Rete match algorithm approach. Thus these speed-ups are not actually measured relative to the optimal sequential implementation. Together these facts about past research account for the relatively large speed-up of 5.1 when compared to the upper-bound value of 9.53 produced by STEPPS.

6.9 Evaluation of STEPPS Performance

STEPPS performed according to requirements during a majority of the testing that supports this investigation, however its limitations with respect to performance (time and space) and static data representations did present some problems. The STEPPS system ran slowly while parsing production systems applications with complex rules such as Tourney and Rubik. However, the system did execute reliably most of the time allowing it to be run as a background task on multi-tasking systems. During the testing process, the only problems that caused STEPPS to halt prematurely were caused by the following:

- The user (me) did not correctly specify the maximum number of rules that could be encountered while processing the application source file.
- A rule name, object class name or variable name was too long for the STEPPS static low-level data representations.
- The STEPPS system could not obtain sufficient memory to complete the processing requirements.

This static data representation problem could often be easily corrected by abbreviating the name of the rule, object class or variable name; however, this "hand-done pre-processing" was sometimes very tedious. The memory problem was successfully addressed by carefully analyzing "high memory turnover" code components and ensuring that the maximum amount of allocated memory was reused. Suggested improvements to STEPPS might involve improving system time performance, incorporating variable length low-level data structures for rule representation and improved memory management.

6.10 Synopsis of Investigation Results

Testing performed during this investigation confirms the findings in the summary of Chapter 4 that the available parallelism metrics developed for STEPPS can only predict the **upper-bound** parallelism in a given production systems application. This conclusion is evident when comparing the level of parallelism achieved using different parallel production system shells with the evaluation of parallelism derived by STEPPS. Some of the comparisons of STEPPS evaluations to actual implementation results are:

- CLIPS monkey and bananas, production parallelism: STEPPS: 3.68 actual: 1.5
- Tourney, node parallelism: STEPPS: 13.5, actual: 3.0, simulation: 8.0
- Rubik, node parallelism: STEPPS: 135, actual: 12.5/13.0
- RAV, production parallelism: STEPPS: 1.548 actual: 0.68

Of the above results, Rubik may be the most accurate because the high level of node parallelism as derived by STEPPS indicates that speed-up should be linear for 100 or

more processors; this implies a speed-up of almost 13 ("almost" accounts for the affect of sequential, as opposed to parallel, rule selection phase) when using 13 processing elements which is the value achieved by the CMU implementation. The remaining result comparisons are not even close in contrast with the Rubik results.

This research cannot make any more than an initial assessment of the characteristics of production systems that exhibit a high degrees of available parallelism due to several significant limitations noted above. The basic characteristics of importance, with regard to available parallelism appear to be:

1. The degree of coupling between rules in the application, also known as the average number of rules affected by executing a given production rule; generally, the tighter the coupling between system rules, the better the chances for increasing parallelism.
2. The average number of working memory actions specified on the RHS, the higher the number of actions, the better the potential for increased parallelism; generally, the greater the number of RHS actions, the greater the potential for increasing parallelism.
3. The number of unique patterns used in constructing the LHS of the production rules; generally, the more unique patterns there are to be processed, the greater the potential for parallelism (particularly node parallelism).

The above three characteristics are typically not totally independent of each other. For instance, a large number of actions on the RHS of a given rule can allow for a large number of different rules to be affected by the execution of that rule. Additionally, increased rule coupling may also result in a significantly increased number of unique Rete patterns.

Of all the applications examined by this research, Rubik may be the best example of a parallelizable production system. Each rule in Rubik affects an average of 70%+ of the rules in the rule base and the dynamics of the system's execution appear to execute the rules with the largest span-of-affect most often. The "heavy" weighting of rules with higher than average span-of-affect causes the upper-bound on production parallelism to be higher than the average number of rules executed. The large number of patterns used on the

LHS of Rubik rules also contribute to a very high level of node parallelism. However, the overall structure of Rubik does not (appear to) conform to the structure typically seen in complex expert systems, such as the RAV system also studied. Significantly more research is clearly required in order to determine the feasibility of applying the techniques used by Rubik to different types of intelligent control problems.

6.11 Summary

The investigation contained in this chapter meets all 3 of the basic objectives discussed in Section 1 of this chapter. In processing the eight production system applications contained in this chapter, the STEPPS system demonstrates that it meets the initial system requirements defined in Chapter 5, Section 2. The results of using STEPPS to evaluate the possible level of parallelism in the eight example applications are also significant. This investigation also reaffirms one of the primary findings of Chapter 4; that the metrics used by STEPPS can only define an **upper bound** on the level of parallelism in a specific application. This is evident when reviewing the level of parallelism for a specific application as predicted by STEPPS with the actual parallelism achieved when implementing the same application using a parallel production system shell. The results of the investigation also support the assertion that the level of rule base coupling is the most significant indicator of the level of available parallelism in a given production systems application. The only significant shortfall of this investigation is that the applications actually examined represent relatively small applications for the most and may not accurately reflect the results of a study involving only larger sized systems. Future research involving the investigation of the level of parallelism in larger production systems applications should be considered.

VII. A Detailed Investigation of Agenda Parallelism

7.1 Introduction

The concept of agenda parallelism is not a new one, many researchers have offered variants of the parallel rule execution model as an extension of the traditional production system paradigm (48) (77) (62) (22). Previous research concentrates mostly on proving the general correctness of executing selected rules in parallel and offers only a few, pedagogical problems which serve to demonstrate the behavior associated with parallel rule execution. Preserving the correctness of the production system paradigm is of paramount importance, therefore much of this past research is vital to establishing the concept of agenda parallelism. On the other hand, much of the past research does not consider many of the practical aspects vital to actually implementing a production system shell capable of efficiently exploiting agenda parallelism. This chapter expands on the initial concepts of agenda parallelism first introduced in Chapter 4. This expanded presentation begins by proving the key assertion in the agenda parallelism approach; that is, the assertion that all non-interfering rules can be executed in parallel without adversely affecting the correctness of the production system paradigm. The ensuing discussion covers the key concerns related to actually implementing a production system shell that effectively and efficiently uses agenda parallelism. The discussion of implementation details surrounding agenda parallelism leads directly to a description of Multi-CLIPS, a production system shell capable of executing multiple rules during the same production cycle. Despite the significant research in this area (cited above), there are currently no other production system shells capable of executing multiple rules during the same production cycle¹ Finally, this chapter extends the original focus of this research effort (static analysis) by providing a limited dynamic analysis of the agenda parallelism in three different production system applications.

¹The one exception is Merit Technology's METE shell discussed in Chapter 4, Section 6.1; however this shell automatically assumes that all rules are non-interfering.

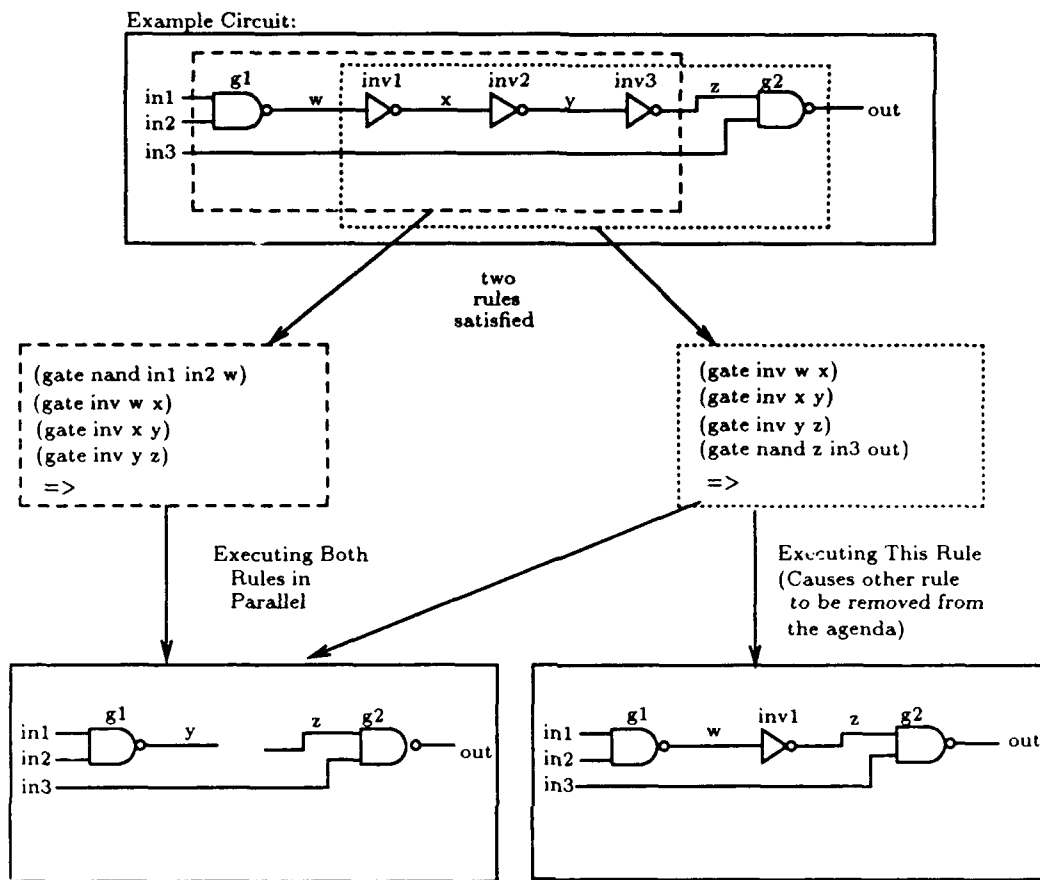


Figure 7.1. An Example of Interfering Rules in a Production System

7.2 Proving the Correctness of Agenda Parallelism

Probably the first and foremost concern in exploiting agenda parallelism is preserving the correctness of the production system paradigm. The brief introduction to agenda parallelism in Chapter 4 states that correctness can only be preserved through the execution of non-interfering rules, but does not provide any proof of this assertion. This section provides proof that executing non-interfering rules in parallel preserves the correctness of the production system paradigm. Parallel rule execution in production systems is directly analogous to the problem of maintaining the determinate behavior of task systems in multi-processor operating systems. Therefore, it is possible to make direct use of the proof of determinate task systems to prove the correctness of agenda parallelism approach.

Parallel rule execution is only possible when it does not adversely affect the deterministic nature of the production system model. Traditional (Post developed) production systems use non-deterministic methods for selecting the next rule to execute(54:134), however given the same initial sets of data and the same sets of rules, the same answer will always emerge over a number of system "runs". This implies that the initial data set causes a deterministic "path" of rules to be traversed in the search for an answer to the original problem. By introducing parallel rule execution into the deterministic process of single rule execution, it is possible to produce non-deterministic (and incorrect) results. Consider the example problem of digital circuit redesign in Figure 7.1: This example uses one of the circuit simplification rules from the digital circuit redesign production system (previously presented in Figure 6.5) to simplify an example circuit. The example rule "looks" for two inverter gates in series and eliminates them in order to simplify the circuit. In this particular example, the RHS of both each rule interferes with both the RHS and the LHS of the other rule. By applying this example rule to both pairs of inverter gates in the example circuit during the same production cycle, an incorrect circuit results. In the example, executing one rule during each production cycle results in a correct circuit because, the execution of the first rule produces a new circuit that no longer supports the application of the second rule, hence that rule is removed from the agenda without being executed. As a direct result of the first rule's execution, the second rule is removed from the agenda without actually being executed. This example does not provide any support for proving that the execution of non-interfering rules is correct, but it does provide an important illustration of the consequences of not preserving the correctness of the basic production system model. Coffman and Denning note that "the problem of non-determinacy can be solved simply by introducing the proper precedence constraints between previously independent tasks" (19:35).

Coffman and Denning present a multiple task environment to support the discussion and proof of determinate task systems that, can also be used to prove the determinate behavior of parallel rule execution in production systems. The task system environment consists of an ordered set $M = \{M_1, M_2, \dots, M_m\}$ of memory cells, a set of system states that depends on the value of the memory cells $S_k = [M_1(k), M_2(k), \dots, M_m(k)]$ and the

set of all system states can be written as $S = V^m$. The system C is composed of a set of tasks C where each task $T \in C$ has a domain of memory cells D_T that it reads information from prior to execution and range of memory cells R_T it writes information to following execution. Finally, executing a task T is analagous to performing the mapping $f_t : V^d \mapsto V^r$ (19:36). Using this model, Coffman and Denning provide the following definitions: (19:38):

- A task system C is *determinate*, if for any state s_0 , all possible execution sequences depend uniquely on the initial value(s) of s_0 .
- Two tasks T and T' are *non-interfering* (can be executed in parallel) if either:
 1. T is the *successor* or *predecessor* of T' or
 2. $R_T \cap R_{T'} = D_T \cap R_{T'} = R_T \cap D_{T'} = \emptyset$

Using these definitions, the theorem: “Task systems consisting of mutually non-interfering tasks are determinate can be proven by induction on the number of tasks in the system” (19:40). For a system containing only 1 task, the proof is trivial because only one execution sequence is possible (19:39). A system containing N non-interfering tasks can therefore be proven to be deterministic using induction from a provably determinate task system with 1 task and the assumption that a task system with $N - 1$ non-interfering tasks is also determinate.

At this point most readers should be able to perceive fully how the problem of parallel rule execution in production systems fits seamlessly into the model for deterministic task systems, however the following brief comparison is provided for completeness and clarity. Consider a production system P , composed of an ordered set of working memory elements $W = \{W_1, W_2, \dots, W_m\}$ and a set of rules R where $(R \in P)$. The state of the production system at any given time during its execution can be expressed as $S_k = [W_1(k), W_2(k), \dots, W_m(k)]$ Each rule (R) contains a set of LHS conditions R_{lhs} , which must be satisfied prior to executing the rule and a set of RHS actions R_{rhs} , that are carried out if the rule is successfully matched and subsequently chosen for execution. For each rule $R \in P$, $R_{lhs} \subset W$ and $R_{rhs} \subset W$, meaning that each rule “reads” a subset of working

memory elements and then “writes” a set of working memory elements when it executes. The mapping of the determinate task system model to the production system model is as follows:

- $M = \{M_1, M_2, \dots M_m\} \mapsto W = \{W_1, W_2, \dots W_m\}$
- $S_k = [M_1(k), M_2(k), \dots M_m(k)] \mapsto S_k = [W_1(k), W_2(k), \dots W_m(k)]$
- $\{\forall T \in C \parallel D_T \in M\} \mapsto \{\forall R \in P \parallel R_{lhs} \in W\}$
- $\{\forall T \in C \parallel R_T \in M\} \mapsto \{\forall R \in P \parallel R_{rhs} \in W\}$

This mapping means that the ordered set of memory cells in the task system are equivalent to the elements in the working memory of a production system and the state of working memory corresponds directly to the state of the production system as the state of the memory cells correspond to the state of the task system. The tasks in the system parallel the rules of a production systems as each rule possesses a domain of working memory elements in the form of the LHS (MATCH) and a range of working memory elements in the form of the RHS (ACT). This mapping of production systems to determinate task systems provides a proof for the assertions originally made in Chapter 4, Section 4. Given that the underlying concepts of parallel execution of non-interfering rules is correct, it is now vital to consider issues closer to the actual implementation.

7.3 Implementation Issues in Agenda Parallelism

As the previous section shows, proving the correctness of parallel rule execution is the original overwhelming concern, however there are a number of significant issues closer to the actual implementation that must also be addressed. Briefly, the three most important issues related to implementing agenda parallelism are:

1. Minimize the affect of parallel rule firing on the sequential conflict resolution strategy of the expert system shell.
2. Maintain the efficiency of the match and act phases and minimize the overhead associated with detecting interference between rules on the agenda.

3. Maximize the amount of parallelism available through parallel execution of non-conflicting rules on the agenda.

Given the complexity of modern production system shells, not affecting the "resident" conflict resolution strategy in some manner is probably not an option; parallel rule execution is very likely to affect conflict resolution in some way. As shown in Chapter 4, agenda parallelism does not increase the execution speed of sequential production system shells, therefore any computation required to detect interference between rule on the agenda must be considered computation overhead. Methods for choosing the set of rules to execute also represents computation overhead because it is not required under the sequential implementation. Because they constitute computational overhead, methods for detecting conflicts between rules on the agenda and choosing rules to execute in parallel must be as efficient as possible. Clearly, intelligent trade-off decision regarding each of the three above issues is necessary to implement a production system shell capable of efficiently exploiting agenda parallelism.

7.3.1 Modifying Conflict Resolution Although conflict resolution in the original Post production system paradigm is non-deterministic, most commercially and academically available shells use some "semi-deterministic" means of choosing a single rule from the conflict set to execute. Some shells such as OPS-5 use somewhat subtle and obscure conflict resolution strategies known as LEX and MEA that favor execution of the more complex satisfied rules (14:62-65). Other shells like ART and CLIPS use somewhat more explicit conflict resolution approaches based on rule priorities stated by the programmer prior to execution ² in addition to favoring more recently satisfied rules (32:453) (5:193). The common element among the conflict resolution strategy of most production system shells is the *leading edge* approach to rule execution. That is, with all other factors being equal, the rule satisfied most recently during execution is the rule chosen for execution (32:453) (14:66). The concept of executing multiple rules without regard for the order in which they were satisfied might appear to totally violate the leading edge approach to

²The (declare (salience <value>)) construct defines the priority of a rule in both CLIPS and ART

conflict resolution; however, this does not appear to be the case because expert system programmers tend to rely on more explicitly defined control methods.

Executing multiple rules from various points on the agenda can produce correct system performance provided that the application is not overly reliant on the leading edge rule execution process to provide deterministic results. The leading edge effect is used most often to merely help the conflict resolution process and is not typically the exclusive basis for conflict resolution (49:134). Specific declarations of rule importance and the use of control rules are more reliable means for ensuring correct program execution (18:95). This observation agrees with Brownson's use of context rules (14:146-150) and explicit rule sequencing (14:209-211) to aid in controlling program execution of applications written in OPS-5. Based on research by Jackson (49), Giaratanno (18) and Brownstown (14), it appears that affecting the implicit (leading edge effect) as opposed to the explicit (declared rule priority) does not adversely affect the deterministic execution of most systems. On the other hand, ignoring the explicit (stated) importance of a rule appears to be much more likely to affect the result of a program because the system developer specifically intends a higher priority rule to be executed prior to (and not in parallel with) rules of lower priority. Therefore, the resolution strategy chosen for the agenda parallelism implementation discussed later in this chapter essentially ignores the leading edge effect, but retains the method of conflict resolution based on explicit priority declaration of selected rules.

7.3.2 Detecting Interference Between Rules Given an agenda of candidate rules for execution, the first step is to actually determine which satisfied production rules conflict with each other. Detection of interference between eligible rules can be accomplished at compile-time or at run-time; alternately, a combination of compile-time and run-time interference detection may be used (62:149). Clearly compile-time techniques promise the most potential for reducing run-time computation overhead because most of the interference detection is performed "off-line". However, compile-time techniques are not very flexible and do not typically allow the maximum amount of parallelism to be extracted (62:149). Run-time rule interference detection is likely to be more expensive in terms of computational overhead, but it also supports extraction of the maximum amount of parallelism available

(62:149). Combining compile-time and run-time rule interference detection techniques is likely to allow the maximum amount of parallelism while reducing the computational overhead costs associated with the run-time approach. However, the combined technique is also more complex than both the compile-time and run-time techniques simply because it combines the two.

A significant amount of past research concentrates on compile-time detection of interference between rules on the agenda despite its significant shortcomings (48) (77) (22). Given a production system application that contains only constant elements (i.e. no variables) in the rule base ³, compile-time techniques can successfully identify all instances of rule interference (62:149). Most researchers engaged in compile-time interference detection can use a priori information concerning possible rule interference to partition the rule-base between several interconnected processors. If the rule base is properly partitioned, each processor is capable of executing rules in parallel in an asynchronous manner (22:2) (48:572). However, compile-time techniques cannot accurately identify non-interfering rules when rules with variable terms are allowed. The most significant shortcoming of compile-time techniques is their ability to only identify rules that cannot possibly conflict with other rules when applied to applications that use variables; in most cases, compile-time methods produce only very conservative results with regard to sets of non-interfering rules (62:149). Therefore, although compile-time interference detection techniques are potentially very time efficient, they are not as likely to allow the large degrees of agenda parallelism possible when using run-time interference detection techniques.

Run-time techniques for detecting interference between rules are likely to add considerably more computational overhead than compile-time techniques, however, run-time techniques appear to be better suited for the large majority of production system applications which use variables. By waiting until the eligible rules with variable are actually instantiated, interference between rules is detected only when interference actually exists not just when it *could* exist. Compile-time techniques could be used to identify when testing between two rules is not required (this would occur only when two rules *could not* conflict), but in the worst case, adding compile-time interference detection actually

³See (22) for two simple example applications

increases the computational overhead. The worst case occurs whenever interference could exist between all rules on the agenda. In this case, rule interference tables generated at compile-time would first be consulted to determine which rules could not conflict. Consulting the compile-time table provides no valid information on existing conflict, therefore the entire run-time interference detection routine must be executed as well. In the vast majority of production system applications, where variables are involved, the run-time technique for detecting interference between rules allows for a greater potential for agenda parallelism than compile-time techniques. Because run-time interference detection appears more compatible with the use of variables and promises the maximum degree of parallelism, it is the method of choice for detecting rule interference in the implementation discussed in the next section. Compile-time techniques are not considered further in this particular implementation for the reasons stated above, but it appears to be a topic worthy of more in-depth study.

7.4 Choosing a Set of Rules to Execute

In order to obtain the maximum possible benefits from agenda parallelism, it is necessary to define an approach to rule selection that yields the maximum available parallelism. One approach is simply to find the largest set of non-interfering rules to execute, a problem that can be stated very simply in terms of a combinatorial⁴ optimization problem:

Maximize:

The Number of Rules Executed.

Subject to:

None of the Rules Executed Interfere.

However, this approach may not always guarantee the maximum amount of parallelism. This is particularly true when agenda parallelism is combined with another, lower-level

⁴This assertion is supported by discussion later in this chapter

decomposition approach such as production parallelism or node parallelism. As an alternative to executing the maximum number of rules, the maximum amount of parallelism may actually be achieved by solving another combinatorial optimization problem:

Maximize:

The Number of Rules Affected by the Execution of the Selected Rules.

Subject to:

None of the Rules Executed Interfere.

This second optimization is more complex than the first problem but it has significance in that it defines the set of non-interfering rules that will maximize the span-of-affect. That is, executing the selected set of rules will affect the maximum possible number of other rules which is not the same as actually executing the largest set of rules.

Both of the problems described in the previous paragraph are cast in the form of optimization problems, therefore it should not be a surprise that both of these problems are *NP-Complete*. Using an *NP-Complete* process to select a subset of the agenda to execute is clearly not a practical option for implementing an efficient production system shell! although the size of the average conflict set in most applications is generally small (48:572, (62:149), extreme cases containing a large number of rules may require significantly more processing overhead than can be recouped through the use of parallelism! Section 2 of this chapter originally acknowledge that maximizing the level of attainable parallelism is very important, but using an *NP-Complete* algorithm to choose a set of non-conflicting rules to execute clearly represents an extreme position. Due to the computational complexity of the optimum solution, an alternative greedy approach to solving each optimization problem must be considered. Greedy methods typically possess polynomial time complexity as opposed to exponential time complexity and although they do not guarantee optimality many greedy algorithms often produce optimal answers for relatively small problems (10:24). This remainder of this section expands upon the two different approaches to

maximizing agenda parallelism outlined previously and also presents a greedy approach to maximizing agenda parallelism based on the first approach.

7.4.1 Maximizing the Number of Rules Executed The first method for maximizing agenda parallelism attempts to select the largest set of non-interfering rules currently on the agenda. Originally this problem is described as:

Maximize:

The Number of Rules Executed.

Subject to:

None of the Rules Executed Interfere.

The first step in solving a problem typically involves finding an appropriate representation; in the case of this problem an undirected graph provides a means for effectively representing the problem. For a graph representation of this problem, the vertices of the graph can be used to represent the eligible rules and the links between the vertices can be used to represent interference between two different rules (vertices); the details of this problem representation are expanded in the following paragraphs. By redefining the number of rules as the cardinality of a selected set of rules and redefining non-interfering as independent, this problem can now be stated as:

Maximize:

The Size of the Rule Subset.

Subject to:

All Rules in the Subset are Independent.

The above problem is *NP-Complete* because in an abstract sense, it meets the 2 requirements for *NP-Completeness*:

1. Exponential Time Complexity: Finding the maximal independent subset of a set potentially requires the examination of all possible subsets of a given set which has an exponential worst case time complexity of $O(2^n)$ where n is the number of rules in the set.
2. Mapping to a known *NP-Complete* problem in Polynomial Time: The problem can be expressed in terms of finding the Maximal Independent Vertex Set (MIS), a problem can be mapped into the Set Covering Problem (SCP) in polynomial time (16:49).

The following discussion presents an example of choosing the largest set of production rules using the SCP technique.

As discussed in the previous paragraph, before solving for the maximum independent rule set, the problem must first be defined in terms of a graph; the reader may notice that the procedure for defining the graph is essentially the same as the process of run-time rule interference detection. In the case of the graph definition problem, the set of rules on the agenda as an undirected graph:

$$\overline{G} := (X, \overline{A})$$

where the set of vertices (X) represents the rules on the agenda:

$$X := \{X_1, X_2, \dots, X_n\}$$

and the set of links between vertices (each represented by the pair of vertices the link connects) \overline{A} represents the conflicts between rules on the agenda:

$$\{\forall i \in 1..n, \forall j \in i..n : (X_i, X_j) \in \overline{A} \text{ iff } X_i || X_j \wedge i \neq j\}$$

where:

$$X_i || X_j \equiv ACT_X_i \cap ACT_X_j = MATCH_X_i \cap ACT_X_j = ACT_X_i \cap MATCH_X_j = \emptyset$$

For example, if rule X_1 interferes with rule X_j , then the vertex pair (X_1, X_2) exists in the set of links for the graph. Because the graph is undirected, the vertex pair (X_1, X_2) could

be written as (X_2, X_1) . However, the limit imposed on the index of the second variable in the problem definition prevents redundant entries in the link set \bar{A} . Using the graph definition of the agenda, it is now possible to demonstrate how the problem of choosing a set of rules that maximizes agenda parallelism can be solved using the MIS and SCP techniques.

Before proceeding with an example of how the SCP technique can be applied to the problem of maximizing agenda parallelism, it is important to understand the relationship between the MIS and the SCP. Two different ways of solving the MIS problem exist (16:49):

1. Generate all of the possible independent sets and then choose the one with the largest cardinality or;
2. Consider the problem as an SCP where the columns of the T matrix ⁵ represent the vertices in the graph and the rows represent links in the graph where $T_{i,j} = 1$ if the vertex x_i is a terminal vertex of link a_j and 0 otherwise. Using this T matrix to solve the SCP, the solution set of columns (\hat{X}) represents the minimum solution to the SCP and the set of all columns (X) minus the columns in the SCP solution (\hat{X}) is the MIS of the graph. The reason that $X - \hat{X}$ is the MIS solution is that $X - \hat{X}$ represents a set of vertices that must be independent and this set of vertices is maximal because the set of vertices in \hat{X} is minimal (covering) (16:49).

The second alternative of the two approaches above provides a somewhat more interesting approach to solving the problem at hand, therefore the following example uses the SCP-MIS approach as opposed to the more "brute-force" approach of the first alternative.

The theoretical aspects of the MIS and SCP problems and solutions have already been well explored, therefore this section concentrates on how these techniques can be used to solve a specific example of the problem at hand. Consider the following agenda of five production rules (all with the same explicit priority) where all LHS and RHS predicates are represented by positive integers:

⁵The T (also known as A^T) matrix is normally the transpose of the adjacency matrix with all diagonal elements set to 1 (16:39); however, in the context of this problem, the definition is altered.

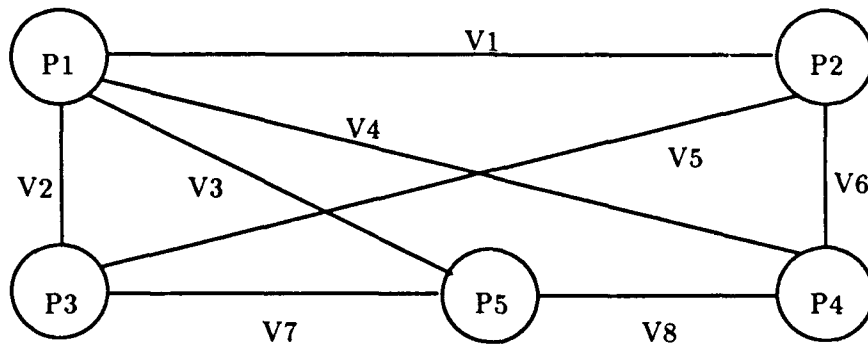


Figure 7.2. Visual Representation of the Example Problem

$$Q = \begin{vmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{vmatrix}$$

Figure 7.3. Adjacency Matrix for the Example Problem

P1: 1, 2, 7, 8 => 3
P2: 1, 7 => 3, 5
P3: 3, 4, 8 => 1
P4: 3, 4, 5, 7 => 2, 7
P5: 6 => 4, 6, 8

This agenda can be represented by the graph $\bar{G} := (X, \bar{A})$ where:

$$X = \{P1, P2, P3, P4, P5\}$$

$$\bar{A} = \{(P1, P2), (P1, P3), (P1, P4), (P1, P5), (P2, P3), (P2, P4), (P3, P5), (P4, P5)\}$$

$$T = \begin{array}{c|ccccc} & P1 & P2 & P3 & P4 & P5 \\ \hline V1 & 1 & 1 & 0 & 0 & 0 \\ V2 & 1 & 0 & 1 & 0 & 0 \\ V3 & 1 & 0 & 0 & 0 & 1 \\ V4 & 1 & 0 & 0 & 1 & 0 \\ V5 & 0 & 1 & 1 & 0 & 0 \\ V6 & 0 & 1 & 0 & 1 & 0 \\ V7 & 0 & 0 & 1 & 0 & 1 \\ V8 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Figure 7.4. The "T" Matrix for the Example Problem

The visual representation of this graph is shown in figure 7.2 and the adjacency matrix Q is shown in figure 7.3. Given this information about the problem, it is now possible to recast it in the form of an SCP problem.

Before applying the SCP search technique to find an optimal solution for this problem, it is necessary to generate the T matrix and then the SCP *tableau*. Using the procedure defined in the second MIS solution technique, it is possible to construct the T matrix shown in Figure 7.4 from the visual graph representation in Figure 7.2. Using this T matrix, the SCP *tableau* in Figure 7.5 can be constructed to facilitate the efficient search for an optimal solution. Essentially, this tableau orders the problem so that the most likely paths to an optimal solution are explored first (16:41). The SCP tableau is constructed in a series of "blocks"; each "block" contains only the vertices that serve to cover the link that corresponds to the block number. For instance, the first block in Figure 7.5 contains only P1 and P2 because these are the only two vertices that serve to "cover" link V1. Logically, the tableau contains the same number of "blocks" as there are links in the graph. The locations of the tableau that are not filled in correspond to "don't care" values. The reason for these "don't care" values becomes evident during the discussion of the SCP search process in the following paragraphs.

To complete the SCP tableau construction, the "cost" associated with adding each vertex to the covering vertex set must be incorporated. By convention, the cost associated with a given vertex appears in the last row of the tableau. In the case of this problem type,

	P1	P2	P1	P3	P1	P5	P1	P4	P2	P3	P2	P4	P3	P5	P4	P5
V1	1	1														
V2	1	1	1	1												
V3	1	1	1	0	1	1										
V4	1	0	1	0	1	0	1	1								
V5	0	0	0	1	0	0	0	0	1	1						
V6	0	0	0	0	0	0	0	1	1	0	1	1				
V7	0	0	0	1	0	1	0	0	0	1	0	0	1	1		
V8	0	0	0	0	0	1	0	1	0	0	0	1	0	1	1	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 7.5. SCP Tableau for the Example Problem Using Unity Costs

each vertex assumes a unity cost. Christofides fails to explicitly mention that unless unity costs are assumed, the correct MIS may not be found when using this method⁶! A proof of this assertion by counterexample is straightforward given a cost of 4 for P1 and a cost of 1 for all other vertices. As a special note, this adaptation of the SCP determines the minimum covering **link** set for the particular problem, not the minimum covering **vertex** set; however this behavior is correct as the set of vertices in the graph minus the set of vertices in the minimum link cover is actually the MIS (16:49).

Finally, the SCP tableau in Figure 7.5 is used to implement the search shown in figure 7.6. Each "level" in the search tree uses a new "block" from the SCP tableau to ensure the T-matrix link corresponding to the current level in the tree is covered. For instance, at the first "level" of the graph, either P1 or P2 can be used to ensure that link V1 is covered. As the search proceeds (typically depth-first), new vertices are added to the vertex set and more links are covered. To ensure optimality, every search tree branch with a cost less than the best solution (all links covered) found so far must be examined further. When the cost of a current branch equal or exceeds the cost of the best solution found so far, then an optimal solution cannot exist further down that branch and it is removed from further consideration. The search for an optimal solution to the example

⁶Christofides does however implicitly describe the problem with unity costs in an illustration of the relationships between different graph covering problems (16:50).

problem produces three optimal solutions, of which two are unique:

$$\hat{X} = \{P1, P2, P5\} \text{ and } \hat{X} = \{P1, P3, P4\}$$

Because the MIS is actually $X - \hat{X}$, the MIS solutions are:

$$MIS = \{P3, P4\} \text{ and } MIS = \{P2, P5\}$$

7.4.2 A Greedy Approach for Maximizing Rules Executed Because the SCP solution to maximizing the number of rules executed is *NP-Complete*, an alternative greedy method is required to ensure acceptable time efficiency in cases of extreme problem dimensionality. The greedy method presented in this section does not guarantee an optimal solution, but possesses only polynomial time complexity as opposed to the exponential time complexity of the SCP technique (10:54). In abstract terms, the greedy algorithm to solve the MIS problem can be expressed as:

1. For every vertex, determine the outdegree (number of links to other vertices in the set), requires $O(n^2)$ time.
2. Order the vertices by the outdegree, vertices with the smallest outdegree first, vertices with the largest outdegree last, requires $O(n \log_2 n)$ time.
3. For each vertex i from first to last, add vertex i to the MIS iff it does not have a connection to a vertex already in the MIS, requires $O(n^2)$ time.

Later in this research process, it was discovered that this algorithm is conceptually similar to, but not exactly the same as, the approximate graph coloring algorithm Christofides attributes to D. C. Wood (16:72). Even though the greedy MIS technique does not guarantee optimal results, it produces optimal results for most small size problems; a counterexample for this greedy algorithm required a graph with 25 vertices! The following example illustrates how the greedy MIS algorithm produces an optimal solution for the original example problem in Figure 7.2.

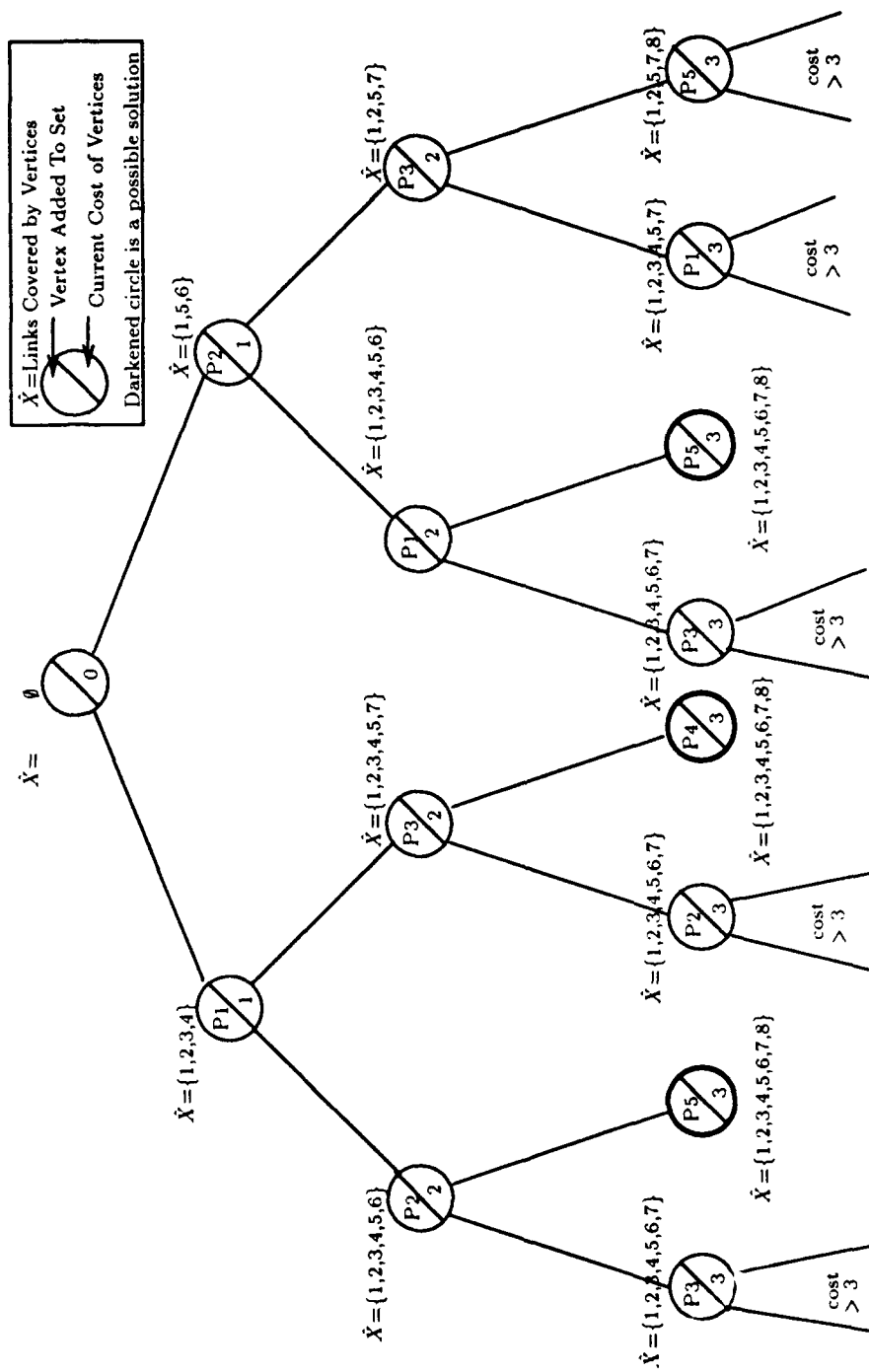


Figure 7.6. SCP Search Tree Generated for the Example Problem

For the original example problem, the greedy MIS algorithm provides a much more straightforward means for finding an optimal solution. In the example problem (figure 7.2), the outdegree of each vertex is:

P1 : 4 P2 : 3 P3 : 3 P4 : 3 P5 : 3

In order, this vertex set is:

P2 : 3 P3 : 3 P4 : 3 P5 : 3 P1 : 4

Given the above ordering of vertices, the third step of the algorithm proceeds as follows:

1. Choose vertex P2, successful no other vertices in the MIS.
2. Try vertex P3, fails because P3 is connected to P2.
3. Try vertex P4, fails because P4 is connected to P2 also.
4. Choose vertex P5, successful because P5 not connected to P2 .
5. Try vertex P1, fails because P1 is connected to P2 and P5.

Thus, the greedy solution to the MIS example problem is $\{P2, P5\}$ which is also one of the optimal solutions to this problem. As shown later, this simple problem is more complex than the ones typically encountered while executing several production system applications. This limited evidence may suggest that the greedy method is sufficient to provide maximum parallelism in all but the most extreme cases and that the SCP approach is not really computationally practical on-line.

7.4.3 Maximizing the Size of The Affect Set In Chapter 3, Section 5.5, the discussion on combining agenda parallelism with production parallelism advocates maximizing the number of rules affected by parallel rule execution, not just the maximum number of rules executed. For example, consider what would happen if each of the rules in the example problem affected different numbers of rules:

P1 : 6 P2 : 3 P3 : 2 P4 : 2 P5 : 2

In this case, executing rule P1 as would affect 6 other rules while executing rules P2 and P5 in parallel would affect 5 rules and executing rules P3 and P4 in parallel would affect only 4 other rules. In the context of this new approach to maximizing agenda parallelism, executing only one rule (P1) provides the optimum solution instead of executing the rules in either of the MIS solutions in parallel! Clearly this second problem is more complex than the first one because the relationships between rules and the types of rule instantiations that make up the agenda must be considered in addition to ensuring the selected set of rules is independent. Also, because this problem has yet to be explored in detail, no greedy algorithms for finding an approximate problem solution have yet been formulated. Due to the substantial complexity of this second optimization problem and the current lack of a greedy method for finding an approximate solution, this section only touches on the basic concepts of this alternate approach instead of providing a detailed discussion of the solution.

Like the first optimization problem, this second problem is also *NP-Complete*, however it is significantly more difficult to actually demonstrate the two vital aspects of *NP-Completeness*. Initially, this problem was defined as:

Maximize:

The Number of Rules Affected by the Execution of the Selected Rules.

Subject to:

None of the Rules Executed Interfere.

Like the previous combinatorial optimization problem, an undirected graph provides an effective representation of the problem; the vertices of the graph represent the rules on the agenda and the links represent interference between pairs of rules on the agenda. The primary difference between this problem and the previous one is that the optimization measure is not as simple as the cardinality of the selected subset of rules. Given the above description and the problem representation specifics, the problem can be defined as follows:

Maximize:

The Subset of Rules that Produce the Maximum Affect Set

Subject to:

All the Rules in the Subset are Independent.

The rationale for exponential complexity again rests on the fact that, in the worst case, all possible subsets of the set of rules making up the agenda must be examined in order to determine which one produces the maximum affect set. Abstractly, this problem can be mapped to an MIS problem as long as the semantics associated with the term “maximal” are redefined. Assuming the existence of some polynomial-time function \mathcal{F} that can be used to compute the number of rules affected by executing some independent set of rules on the agenda, this function can be used to map this second optimization problem into an MIS problem in polynomial time. Provided the assumed polynomial-time function \mathcal{F} actually exists or can be derived, this problem mapping in concert with the exponential time complexity argument above is sufficient to prove the *NP-Completeness* of this second optimization problem. Following is a high-level description of the hereto hypothetical function \mathcal{F} .

As alluded to earlier, solving this second optimization problem requires a good deal more information about the rule base and the agenda than the first problem. To ensure that only a non-interfering (independent) set of rules are chosen for execution, the run-time generated graph matrix (\mathcal{Q} from the previous section) is required. In addition to the graph matrix, the affect-set \mathcal{S} matrix is required to determine which rule types are actually executed by the execution of each given rule type⁷. Finally, each rule in the agenda set represents an instantiation of a specific rule type, this information is needed to “tie” together the separate maximize and subject problems represented by the \mathcal{Q} and \mathcal{S} matrices. A matrix \mathcal{K} can be used to represent what type of rule in the rule base is instantiated to form

⁷See Chapter 4, Section 5.3 for a discussion of affect-set matrix construction

each rule on the agenda, this matrix can be defined as:

Agenda Instantiations:

$$\kappa_{i,j} = \begin{cases} 1 & \text{if rule } i \text{ is an instantiation of rule } j \\ 0 & \text{otherwise} \end{cases}$$

Using all three pieces of information, the \mathcal{Q} , \mathcal{S} and \mathcal{K} matrices, it should be possible to define this second optimization problem as a 0-1 Integer Linear Programming (ILP) problem.

7.5 Implementing Agenda Parallelism Using CLIPS

Multi-CLIPS is a modification of version 4.3 of the CLIPS production system shell which is capable of executing multiple rules during each production cycle. Multi-CLIPS was developed to explore important concerns related to actually implementing agenda parallelism. The Multi-CLIPS implementation relies on the implementation decisions related to conflict resolution, rule interference detection and rule selection outlined in Section 2 of this chapter. Multi-CLIPS uses only explicitly defined rule priority for conflict resolution, the run-time technique for rule interference detection and choosing the largest set of non-interfering rules for maximizing the agenda parallelism. Choosing the largest set of non-interfering rules was chosen as opposed to maximizing the number of rules affected. Although executing the largest set of rules may not yield maximum agenda parallelism, an efficient approximate solution for choosing the largest set of non-interfering rules is available and this method represents a much lesser impact on the original CLIPS architecture than maximizing the number of rules affected. In addition to studying the affect of the previous decisions, the actual implementation brings to light some of the liabilities related to agenda parallelism that are not obvious from the initial discussion of concerns.

The Multi-CLIPS implementation required changes to the CLIPS Inference Engine and Functions subsystems to allow execution of non-conflicting rules during the same production cycle. The changes center around three different actions required to support parallel rule execution:

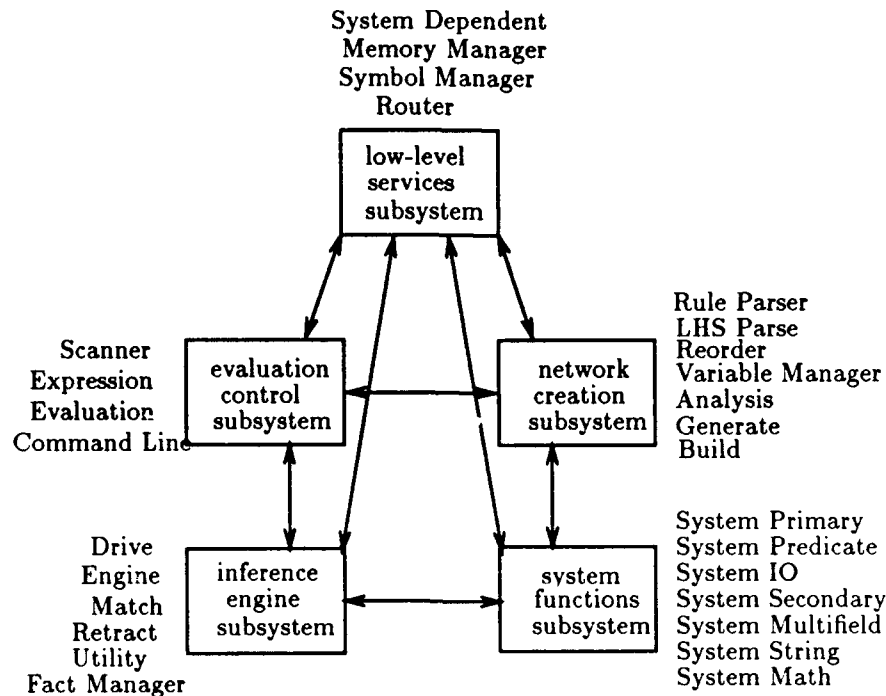


Figure 7.7. The CLIPS High-Level System Architecture

1. Right-Hand-Side Evaluation: Evaluate the RHS of a satisfied rule as soon as it is added to the agenda.
2. Compress the Agenda: Find the largest set of non-conflicting rules and put them at the front of the agenda.
3. Multiple Rule Execution: Divide the agenda into independent parts and execute all possible non-interfering rules.

These three actions required some significant changes to some CLIPS modules and relatively minor changes to others. The most significant changes are to the Inference Engine subsystem and are concentrated in the Fact_Manager and Engine modules. Right-hand-side Evaluation affects both the Engine and Fact_Manager modules (and the System_Secondary module in the Functions Subsystem), but compressing the agenda and multiple rule execution affects only the Engine module. Following are a brief discussion of the changes made to the CLIPS shell and their impacts on its performance.

One of the most complex problems related to finding the set of non-interfering rules involves evaluating the RHS of a satisfied rule when that rule is actually added to the agenda. When the CLIPS system finds a set of variable bindings that cause a given rule to be satisfied, it places the list of variable bindings and the name of the rule on the agenda (17:56). The variables on the RHS of the satisfied rules are not actually instantiated until that rule is selected for execution. Research into the ART and OPS-5 production system shells reveals that this method for deferring RHS variable binding until rule execution is a common practice for high-performance shells (5) (58). Deferring RHS variable binding reduces unnecessary computation that occurs whenever a satisfied rule is removed from the agenda (because the facts in the working memory no longer support the existence of that satisfied rule). However, if all RHS variables are not bound at the time the rule is added to the agenda, it is impossible to determine which facts are to be removed from working memory and what new facts are to be added to working memory. Thus, deferring RHS variable binding does not provide the necessary RHS (act) information for detecting interference between satisfied rules.

To ensure that interference between all satisfied rules is detected and only non-interfering rules are selected for parallel rule execution, early RHS binding of satisfied rules is essential. Multi-CLIPS evaluates and binds all RHS variables at the time that a satisfied rule is placed on the agenda to support the identification of interfering rules. During the evaluation of RHS functions and bindings of RHS variables, functions related to working memory (assert, retract and modify) are "intercepted" just before they are actually allowed to take place. As a result, the RHS predicates are now available, but working memory is not affected. The time complexity for evaluating and binding the RHS variables of a rule at the time the rule is added to agenda is no different from the time required to evaluate and bind the rule when it is executed; however, these time complexities are the same if and only if none of the satisfied rules are removed from the agenda (except by being executed). It is impossible to determine, prior to run-time, which rules will be removed from the agenda and which will not, therefore any order-of difference between sequential and parallel rule execution can only be in terms of number of rules satisfied to number of rules removed from the agenda.

The largest changes to the CLIPS shell involves the eight functions added to the Engine module to 1) detect interference between rules, 2) choose the largest set of non-interfering rules and 3) reorder the agenda and execute the chosen non-interfering rules. Compressing the agenda requires the following four major operations:

1. Determine how many rules on the agenda have the same highest priority (salience). requires $O(Rules)$.
2. Form the Q matrix (see Section 3.2) based on the existence of conflicts between rules, requires $O(Rules^2 \cdot \max(LHS_Size \cdot RHS_Size, RHS_Size^2) \cdot String_Length)$.
3. Use the greedy MIS algorithm to select the largest set of non-interfering rules to execute, requires $O(Rules^2)$.
4. Rearrange the agenda so that the non-interfering rules are at the front of the agenda, requires $O(Rules)$.

$Rules$ represents the number of rules on the agenda, LHS_Size is the average number of predicates on the LHS of rules on the agenda, RHS_Size is the average number of predicate on the RHS of rules on the agenda and $String_Length$ is the average length of a string used to represent each predicate.

In the final phase of the multiple rule execution, all of the non-interfering rules at the front of the compressed agenda are executed sequentially before the next production cycle begins. Based on the number of rules in the largest non-interfering rule set, the agenda is divided into a "hidden" part and a "visible" (or default) part. The operation of dividing the agenda requires only $O(1)$ time because the dividing point is determined by the greedy MIS algorithm. Dividing the agenda makes the execution of multiple rules during the same production cycle essentially transparent to the rest of the CLIPS system. All rules on the hidden agenda can be executed sequentially without affecting operations carried out on the standard agenda. Old rules that are no longer satisfied can still be removed from the default agenda without affecting any of the rules on the hidden agenda. Additionally, adding newly satisfied rules to the default agenda prevents the accidental execution of interfering rules (recall that newly satisfied rules are normally put on the

front of the agenda). Execution of rules on the hidden agenda continues until it is empty at which time the agenda must again be compressed. Due to the difference in the processing time required for different rules (in different applications), there is no accurate order-of for the execution of a number of rules. However, as demonstrated in Chapter 4, Section 4.2, the order-of for executing multiple rules during the same production cycle on a sequential computer is essentially equivalent to executing the same rules sequentially during a number of production cycles.

7.6 *Parallel Rule Execution Case Studies*

The availability of the Multi-CLIPS system dictates that one should look at measuring the level of agenda parallelism in a couple of different production system applications. From the applications measured by the STEPPS system in Chapter 5, the following three have been chosen for measurement of agenda parallelism:

- The "monkey and bananas" intelligent control problems
- A CLIPS adaptation the *bogus.kb* SDI scenario simulation
- The DCR system for local digital circuit redesign/optimization

These three examples are relatively "simple" systems, but they serve to illustrate some of the significant differences in agenda parallelism. Additionally, these systems require no external I/O which is currently a problem for the initial implementation of the Multi-CLIPS system. Because the Multi-CLIPS design represents a prototype system for exploiting agenda parallelism, it also suffers significantly in terms of efficiency when compared to the sequential CLIPS shell. Because of the problems with efficiency, actual performance is discussed only in abstract terms related to time complexities and number of rules executed in each production cycle.

Performance information for each example problems in the case study is presented in terms of the following parameters:

- the number of non-interfering rules executed during a given production cycle

- the number of rules retracted from the agenda without being executed during a given production cycle
- the number of rules on the agenda during a given production cycle

The number of rules executed represents the level of agenda parallelism actually achieved for the application. The other two parameters represent measures of computational overhead introduced by the agenda parallelism approach. Retracted rules represent computational overhead associated with immediate RHS element binding that is not required for sequential rule execution. The number of rules on the agenda provides an indication of the level of processing overhead required for rule conflict detection.

7.6.1 The Monkey and Bananas Problem The “monkey and bananas” intelligent control problem represents an application at the lower extreme of the *agenda parallelism* “spectrum”. During each production cycle, only one new rule becomes eligible to execute and thus is the one chosen and executed. This yields a net result of no agenda parallelism realized. For readers familiar with this particular production system problem, this tends to make sense. The actions and decisions of “the monkey” are the only action that the problem is concerned with, and since “the monkey” only concentrates on solving the problem one step at a time, a sequential path of trial and error actions takes place. The strictly sequential problem solving actions used in this problem, therefore, result in a single thread of control and no possibility for applying agenda parallelism. For this problem, no visual representations of performance are provided because no agenda parallelism was realized and neither of the overhead factors had any effect on the results.

7.6.2 The SDI Scenario Production System The Bogus.kb application is part of this initial case study because it was designed by Merit Technology with parallel rule execution in mind (7:88), and therefore, (potentially) contains significant amounts of inherent agenda parallelism. Surprisingly, the Bogus.kb system, as received, is not capable of realizing any agenda parallelism using the approach outlined earlier in this chapter. The reason is that Bogus.kb was designed for executing all rules in parallel during each production cycle without regard for possible rule interference. Moderate modifications were necessary in

order for Bogus.kb to realize any significant level of agenda parallelism. Achieving the level of agenda parallelism obtained by Merit Technology researchers would require abandoning the most important agenda parallelism edict: only non-interfering rules can be executed in parallel.

Even given the limitations introduced by the design assumptions, the Bogus.kb system is still capable of achieving significant levels of agenda parallelism. The execution traces for the Bogus.kb application in terms of number of rules on the agenda and number of rules executed are shown in Figure 7.8. Because Bogus.kb does not retract any facts, all rules, once satisfied, remain satisfied and are not removed from the agenda. Since no rules are retracted, the only computational overhead is introduced by rule interference detection. Using the Merit Technology benchmark (4 initial facts) test procedure, a 31 production cycle average of 39.7 rules on the agenda and 5.9 rules executed was observed. The average number of rules executed using Multi-CLIPS (5.9) is significantly less than the average of 29.33 when using the METE algorithm (51:195). Although the Multi-CLIPS approach does not produce the same high of parallelism as the METE algorithm (for Bogus.kb), it is more extensible. Based on the experience gained from this research effort, the assumption that rule interference does not exist (the basis for METE) does not appear valid for most production systems applications. Thus, the Multi-CLIPS approach is likely to be the only agenda parallelism alternative for most production systems applications.

7.6.3 The Digital Circuit Redesign Expert System Although it was not specifically designed for it, the DCR system exhibits significant levels of agenda parallelism. However it also brings out well the liabilities associated with this level of parallelism. Figure 7.9 displays the level of agenda parallelism achieved for the DCR system as well as the overheads encountered in terms of rule retracted and rule interference detection. The data in this figure was obtained by running Multi-CLIPS on the initial design of a Booth Recoder Circuit which is just a small part of AFIT's floating point, application-specific processor (FPASP) design. As noted in the system description in Chapter 6, Section 3, the DCR system executes in several sequential "phases". The different phases of execution can be seen clearly on both the "rules executed" and "agenda size" parts of Figure 7.9. When

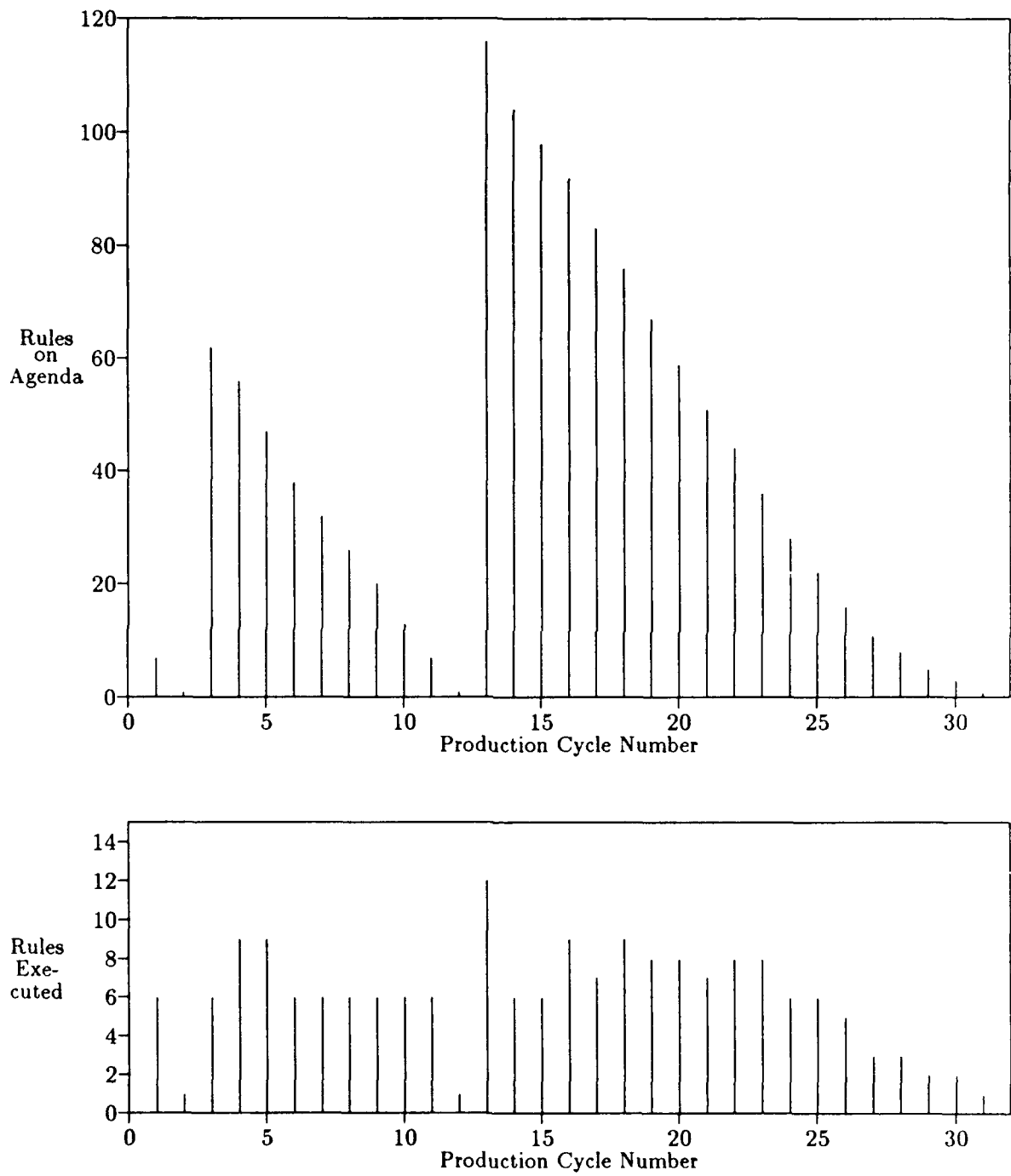


Figure 7.8. Agenda Parallelism Execution Traces for Bogus.kb

each new execution phase begins, the level of parallelism increases significantly and then falls off gradually up until the end of the phase. From the earlier description of the DCR system, the execution phases are: pre-transformation simulation, circuit transformation, post-transformation simulation, evaluation of simulation results and finally print-out of the transformed circuit.

Even though the DCR system possesses a significant amount of agenda parallelism, it also encounters a significant amount of computational overhead. The size of the agenda for some production cycles, particularly those close to the start of a new phase, is large indicating significant processing requirements. Additionally, a number of rules are retracted during the circuit transformation phase which also increases the level of computational overhead. During one particular cycle of the circuit transformation phase, the number of rules retracted from the agenda is actually greater than the number of rules executed. For the 28 production cycle example DCR execution on Multi-CLIPS, a per cycle average of 15.86 rules on the agenda, 13.71 rules executions and 1.64 rules retractions was observed. From these results, it is possible to determine that some interference between satisfied rules typically exists, but that nearly every rule on the agenda is either executed or retracted during a given production cycle. Rule retractions represent an uncontrollable overhead factor. In the case of the example presented here, 46 rule retractions occurred. The number of rule retractions accounts for only a small portion (approximately 10.3%) of all rules satisfied. Despite these overheads, combining agenda parallelism with node parallelism (or production parallelism) could lead to average speed-ups that are far more substantial than either of the two alone.

7.7 Summary

The combination of theoretical discussion and practical experience gained from the implementation and use of the Multi-CLIPS system provides a much clearer picture of the assets and liabilities associated with agenda parallelism. The theoretical basis for agenda parallelism proves that deterministic system behavior can be maintained by executing only non-interfering rules in parallel during each production cycle. Of the possible rule interference detection options, run-time rule interference detection allows for maximizing

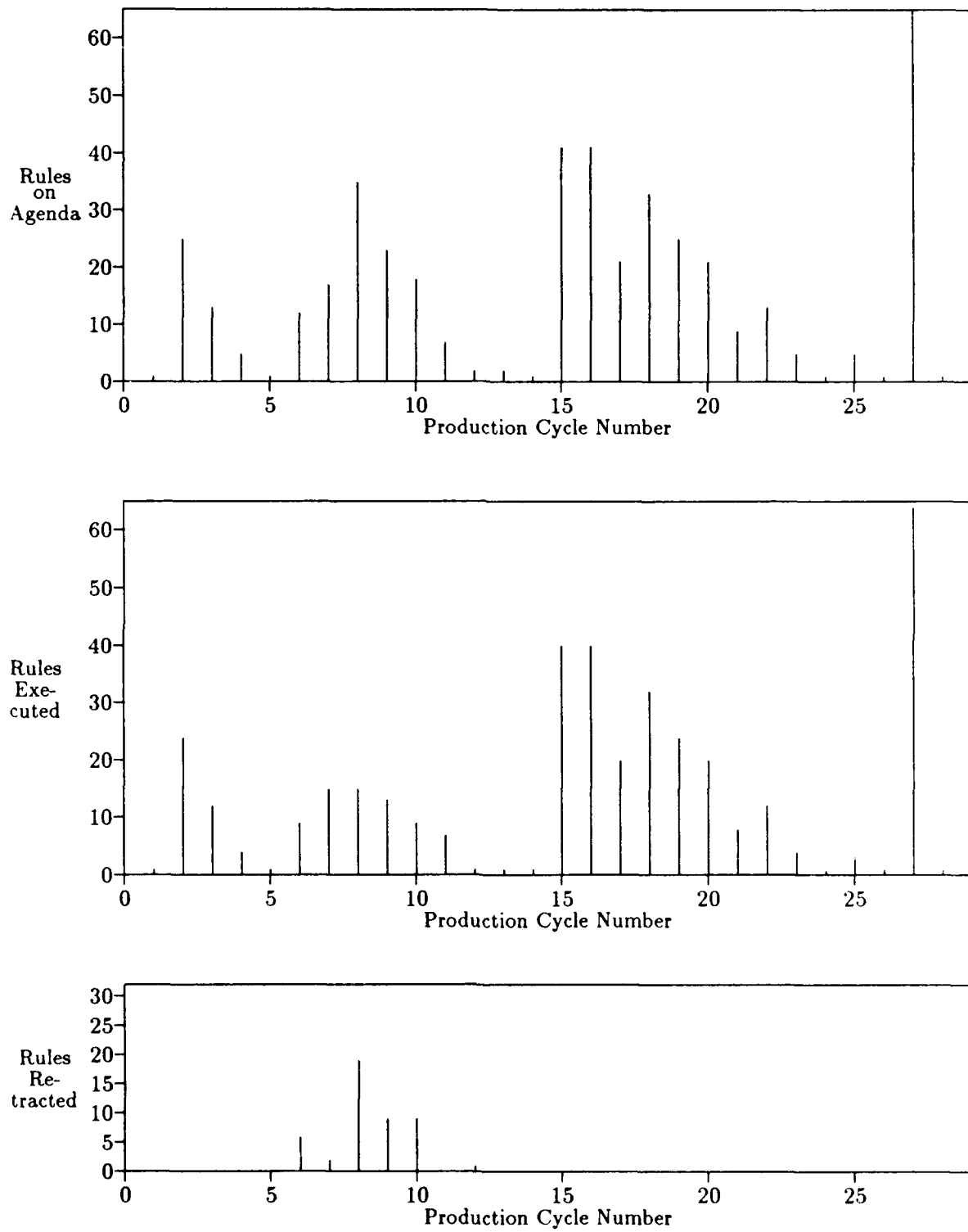


Figure 7.9. Agenda Parallelism Execution Traces for the DCR System

parallelism, but it also exacts a potentially significant overhead cost. The problem of selecting the largest set of non-interfering rules to execute during a given production cycle proves to be an *NP-Complete* problem. Although a polynomial-time approximation to the rule selection problem is available, it does not allow for the largest theoretically achievable agenda parallelism to be exploited. Like rule interference detection, the selection of the set of rules to execute during a given production cycle constitutes another potentially significant computational overhead. Binding of RHS variables at the time newly satisfied rules are added to the agenda (instead of when they are selected for execution) introduces computational overhead whenever rules are removed from the agenda without being executed. Finally, the production systems in the case study demonstrates that agenda parallelism can provide significant increases in parallelism, but that those increases are very application dependent.

The case study of three production system applications performed in this chapter, demonstrates the usefulness of agenda parallelism. Despite the significant amount of average agenda parallelism (rules executed per cycle) in some applications, the level of agenda parallelism can change significantly from cycle to cycle; this observation supports the assertion that agenda parallelism should not be used as an independent decomposition method. Instead of using agenda parallelism as an independent approach, it should be used only to augment lower level decomposition techniques such as production parallelism and node parallelism. As noted in Chapter 4, increases in parallelism available from executing rules in parallel have a potentially multiplicative effect on these lower levels of parallelism. For example, the upper bound on node parallelism for the DCR system application is 21.5 and the average agenda parallelism (from the case study) is 13.7. Temporarily ignoring any potential overheads, a DCR implementation combining node parallelism and agenda parallelism gives an upper bound speed-up of 295. This value is significantly higher than the best upper bound parallelism (using a single decomposition approach) for any of the applications studied in Chapter 6. However, a speed-up of 295 fold does represent an absolute upper bound. The overheads discussed in the previous paragraph and the likelihood that all rule executions cannot be performed completely in parallel⁸ will reduce achiev-

⁸See Chapter 4, Section 6.1 for details on combining node and agenda parallelism

able parallelism to some extent. Despite the adverse affect of computational overhead, combining node and agenda parallelism does provide the possibility of significantly increasing execution speeds when compared to using node parallelism or agenda parallelism exclusively.

Agenda parallelism shows significant promise for increasing the execution speed of production system applications. This is particularly true when combining agenda parallelism with other production systems decomposition approaches such as node parallelism. Combining these approaches into a **multi-level-parallelism** approach provides the potential to multiplicatively increase available parallelism and thus speed-up. Conversely, the potential increases in execution speed are not without costs in terms of both efficiency (computational overhead), and in some cases, risking the correctness of the application by subtly altering the conflict resolution method. Although the information in this chapter addresses many hereto unanswered questions related to agenda parallelism, significantly more research is warranted.

VIII. Conclusions and Recommendations

8.1 Introduction

The closing chapter of this thesis begins by reviewing, from a very high level, the contents of this thesis and how it supports the research goals put forward in the first chapter. Following the review of the thesis, the most prominent findings of this research are presented. Finally, the last section discusses possible directions for future research in the area of parallel production systems.

8.2 Review of Thesis

The overall focus of this research is to obtain a better understanding of the factors that affect parallelism in production systems. This research addresses the broad focus of this investigation using the two-pronged approach illustrated by Figure 8.1. Chapter 2 presents the extensive research into the characteristics and complexities (computational) of production systems. This presentation includes a detailed treatment of the complex match algorithm, Rete, which supports highly efficient sequential production systems implementations. Chapter 3 offers a broad, but concise overview of the concerns associated with increasing the execution speed of problem solving through the use of parallel computing. Chapter 4 ties together many of the concepts from Chapters 2 and 3 by providing an extensive discussion of how parallel computing can be applied to production systems. This chapter also derives metrics for quantifying the level of parallelism in production systems when using various decomposition approaches. It is at Chapter 4 that the research diverges into two related topics:

1. The primary goal of accurately quantifying parallelism in production systems when using different decomposition approaches.
2. The secondary goal of investigating parallel rule execution in production systems. Agenda parallelism represents a decomposition approach that is still relatively unexplored in comparison to most of the decomposition approaches investigated.

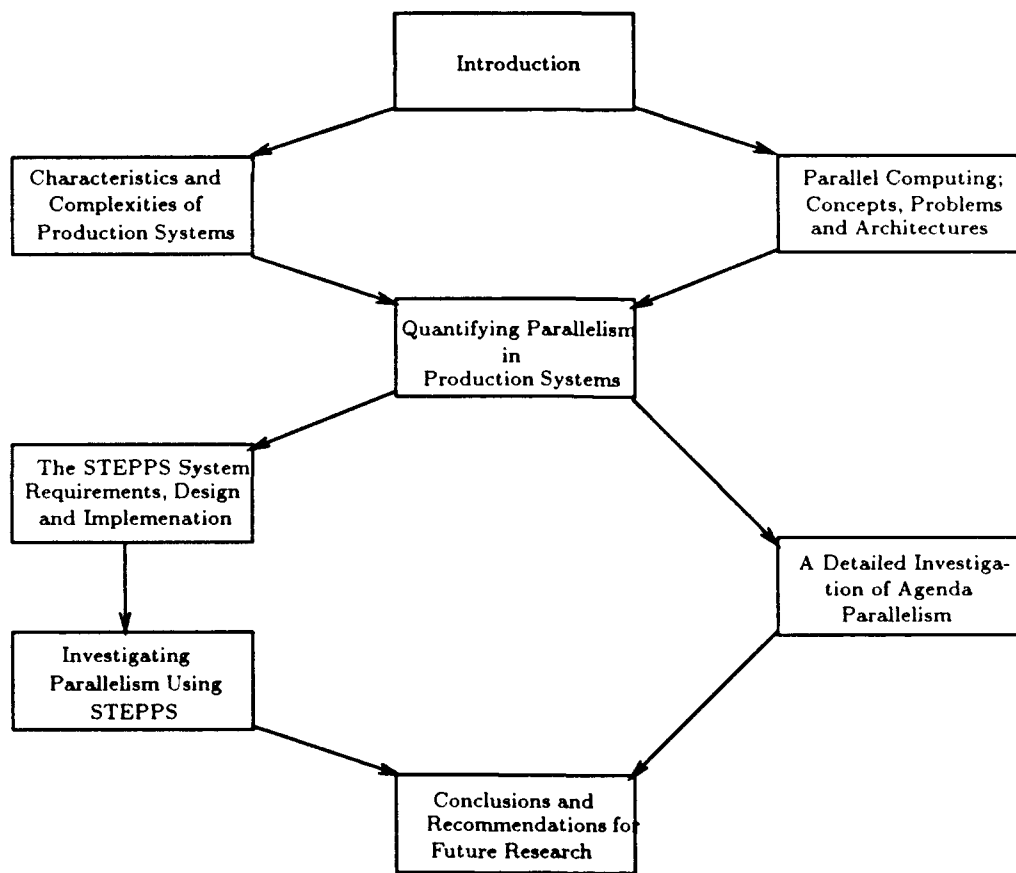


Figure 8.1. A Visual Overview of the Thesis Document

Chapters 5 and 6 detail the development and use of STEPPS, a software tool for evaluating the level of parallelism in different production systems applications. Chapter 7 presents the results of a detailed investigation into the advantages and liabilities associated with parallel rule execution (*agenda parallelism*) in production systems. Finally this chapter reviews some of the most prominent findings from the research and provides recommendations for follow-on research.

8.3 *Research Results*

Because the focus of this research diverges at a critical point, the key research findings are presented in the following separate sub-sections (8.3.1 and 8.3.2).

8.3.1 Measuring Parallelism in Production Systems Accurate measurement of parallelism in production systems from static system characteristics was initially the only goal for this research. The following are key findings related to the investigation of measuring parallelism in production systems:

1. The dynamic nature of production systems (the rule base is the only static item) prevents the accurate determination of parallelism in production systems when using static analysis techniques. Some decomposition approaches allow a rough upper bound on the level of parallelism to be evaluated, but other decomposition approaches allow no such evaluation:
 - (a) *Application Parallelism* - An upper bound on the level of parallelism is determined by the number of separate functional partitions (each with its own processor) that are contained in the application.
 - (b) *Agenda Parallelism* - The number of rules that can be executed in parallel during a given production cycle cannot be determined prior to run-time. Thus, static analysis techniques do not allow for an evaluation of agenda parallelism.
 - (c) *Production Parallelism* - An upper bound on production parallelism is the average number of rules affected by a given rule execution; however, this research

shows that rule-to-processor partitioning may reduce this upper bound parallelism significantly.

- (d) *Node Parallelism* - An upper bound on node parallelism is the average number of unique Rete network nodes affected by the execution of a given rule. The number of different patterns in the Rete network can affect this upper bound parallelism.
- (e) *Intra-node Parallelism* - The number of Rete tokens propagated through the Rete network each cycle cannot be determined through static analysis. Therefore, the level of intra-node parallelism cannot accurately be determined using static analysis techniques.

2. Little correlation between the levels of parallelism predicted by STEPPS and actual results obtained from parallel production systems research is possible. This is due primarily to:

- (a) Very few of the systems available for analysis have been executed using one of the parallel decomposition approaches, and those applications that have been evaluated using a parallel decomposition approach have not been evaluated for more than one decomposition approach.
- (b) STEPPS allows only an estimate on the **upper bound** parallelism when applying a given decomposition approach to a given production system application and does not account for the overhead factors injected by implementation on a particular parallel architecture.

Of the results that can be correlated, the STEPPS estimate of upper bound parallelism is always greater than the level of parallelism that was actually achieved, thus indicating its capability to accurately evaluate **upper-bound** speed-up figures.

3. Of the measurable decomposition techniques, node parallelism appears more promising for realizing significant speed-ups. For many of the measured applications, the node parallelism and production parallelism were approximately equivalent. However, node parallelism represents a dynamic partitioning of the problem as opposed

to the static partitioning of production parallelism. As a result of dynamic problem partitioning, node parallelism may incur higher communication costs. But node parallelism is not as likely to suffer from the significant load imbalance and computational overhead evident in previous production parallelism implementations. Current implementations of node parallelism require closely-coupled parallel architectures in order to minimize inter-processor communication costs.

4. The most successful experiments in node parallelism have been conducted on bus-connected multi-processor systems that are limited to approximately 16 processors. Parallel architectures that use cache-connected busses, shared-memory type message passing techniques, and extensible processor-to-memory routing techniques (such as BBN's Butterfly series computers) may increase this level of parallelism as much as 10-fold. Massively parallel production systems, if they are possible, may require an approach that is dramatically different from this closely-coupled approach. One of the possible areas for further research discussed in Chapter 4 involves mapping production systems to Single-Instruction, Multiple-Data-Stream (SIMD) or Very-Long Instruction Word (VLIW) type architectures.

8.4 *Implementing Agenda Parallelism*

This research significantly extends the theoretical basis of agenda parallelism and provides significant insight into the assets and liabilities associated with actually implementing this level of parallelism. Specifically the key research findings are:

1. The problem of maximizing agenda parallelism while maintaining the deterministic behavior of the system (correctness) is, in actuality, an *NP-Complete* problem. However, case studies indicate that the optimal solution can typically be found using a polynomial-time approximation technique. However, this is not meant to infer that an optimal solution to the problem will always be found by the approximation technique.
2. Agenda Parallelism has a potentially **multiplicative** affect on other levels of parallelism such as production parallelism or node parallelism. This means that agenda

parallelism can be effectively combined with either of these other decomposition approaches because it does not interfere with them.

3. Implementing agenda parallelism adds computational overhead in the form of:
 - (a) Rule interference detection required to ensure that only rules that do not interfere with each other are allowed to execute in parallel.
 - (b) Selection of a set of non-interfering rules to execute in parallel during each production cycle (see finding 3a.).
 - (c) Immediate RHS variable binding of satisfied rule; realizing that some of these rules will later be retracted without being executed.
4. A small cross-section of applications studied indicates that the amount of agenda parallelism in a given application can vary significantly. Some applications with a "very narrow focus" of control may possess no inherent agenda parallelism whereas applications with "multiple threads of control" may exhibit a significant amount of agenda parallelism.

8.5 Recommendations for Future Research

This research covers the evaluation and measurement of parallelism in production systems when using a number of different decomposition approaches. However, this research does not consider a number of other concerns associated with parallelizing production systems. Some of these areas of concern are:

1. Increased levels of parallelism that can be obtained by adding functional decomposition pipelining between the phases of the production cycle to the primarily match-phase oriented decomposition techniques currently employed.
2. The affect of semantic network knowledge representation on the level of achievable parallelism. Both ART and the upcoming CLIPS Version 5.0 allow the use of semantic networks in conjunction with rule-based knowledge representation.
3. Alternative means for constructing the Rete network such as enhanced binary Rete networks and Rete node splitting techniques proposed by researchers at CMU (74).

The affects of the above areas of concern on the level of available parallelism may be vital to the development of efficient and effective parallel production shells. Therefore, each should be considered for incorporation into an enhanced STEPPS tool. This research shows the relative futility of attempting to accurately predict more than an upper bound on realizable speed-up of production systems when using only static analysis techniques. Dynamic analysis techniques will likely require significantly more development work, but these techniques are required in order to obtain more than an upper-bound measurement of available parallelism.

This research explores many of the more important aspects of agenda parallelism in detail, but leaves an equally large number of questions unanswered. Some of these unanswered questions involve the following:

1. The current version of Multi-CLIPS is a prototype and therefore suffers significantly in terms of performance when compared with the original CLIPS version. Due to the performance problem, it was not possible to accurately determine the effect of interference detection and resolution on system performance.
2. The abbreviated case study for this research uses only a few, relatively small (less than 50 rules) production systems applications. Larger applications need to be evaluated to determine the following:
 - (a) Whether the relaxation of the "leading edge" conflict resolution strategy has a significant effect on the deterministic behavior of larger systems.
 - (b) Whether larger applications possess the significant level of agenda parallelism observed in some smaller systems.

Both of the above future research goals will require the development of a more robust version of Multi-CLIPS than currently exists. Future researchers may wish to use the upcoming CLIPS version 5.0 as a basis for future versions of Multi-CLIPS in order to include an evaluation of semantic network knowledge representation with a detailed analysis of agenda parallelism.

8.6 *Summary*

In closing, this research contributes toward a better understanding of the amount of parallelism can be extracted from production systems. Static analysis techniques can yield only a "best estimate" of available parallelism in production systems applications. Additionally, this "best estimate" is available for only a few of the possible production systems decomposition approaches. The lack of empirical evidence simply does not allow a better estimate of available parallelism when using static analysis techniques. Although this is the case, the investigation provides the important characteristics of applications concerning high degrees of inherent parallelism. Given the flexibility of knowledge representation in production systems, it is theoretically possible for future applications to be "cast in the mold of" applications that exhibit high degrees of inherent parallelism. Agenda parallelism is not a new production systems decomposition approach, but it is one that has remained relatively unexplored. This research shows that agenda parallelism has the potential to significantly (even multiplicatively), increase available parallelism in many applications. However, realizing this increased parallelism requires an astute handling of overheads introduced by this decomposition technique. Significantly more research is required in order to fully understand all of the implications of agenda parallelism both from a correctness and a performance point-of-view.

Appendix A. *An Introduction to Logic Programming*

A.1 *Introduction*

The focus of this research has been on productions systems instead of logic programming, however this appendix provides some basic information on this alternative paradigm for constructing expert systems shells and applications for the sake of completeness. Prolog is a programming language based on first order predicate calculus (54:199) and is an embodiment of the backward chaining approach to expert systems. In the formal sense, Prolog is a language and not an expert system shell like CLIPS or OPS-5, but there are a number of commercial backward-chaining shells such as: M-1 and VP-Expert that use the same control structure as Prolog. However, Prolog allows for the relatively easy construction and explanation of examples that illustrate the backward chaining approach and how it differs from the forward chaining approach described in Chapter 2. This appendix begins with the simple example problem of defining part of a family tree and its relations. Based on this example, the appendix then provides a detailed description of the algorithms that "drive" a backward chaining search process. Finally, the simple example problem is used in a brief discussion of available parallelism in the Prolog language and the constraints on this parallelism.

A.2 *An Example Problem in Prolog*

As its name implies, the Prolog language relies on specifying and solving a problem primarily using logic. This approach requires the programmer to think about the problem in a different light; instead of specifying exactly how the problem is to be solved, Prolog

English	Predicate Calculus	Prolog
implies	\leftarrow	$: -$
and	\wedge or $\&$,
or	\vee or $+$;
not	\neg or \sim	<i>not</i>

Figure A.1. A Comparison of Logic Syntax

places more emphasis on what is to be done(11:6). By interfacing the problem description with the implicit control structure of the Prolog interpreter, a programmer is capable of describing the behavior of a complex procedural program in a very few lines of Prolog code(11:8). However, before Prolog can be used to solve a specific problem, that problem must be correctly specified using a restricted logical syntax. First, Prolog automatically takes any words beginning with an upper-case letter (such as "This" or "X") to be variables and words beginning with a lower-case letter (such as "that" or "x") to be constants. Relationships between variables and atoms are expressed in the form of logical predicates such as:

likes(george, prolog).

which is typically interpreted as "george likes prolog" which explains a relationship (however unlikely) between the person george and a specific programming language. If the above predicate were written:

likes(george, Prolog).

it would essentially mean "george likes (something)" because the word "Prolog" begins with an upper-case letter and thus is a variable and could potentially take on the value of anything that "george likes". To round out this discussion of Prolog syntax, Figure A.1 contains a cross-reference between the required logic symbols expressed in English, predicate calculus and Prolog.

Using the Prolog syntax, a simple problem related to the family tree depicted in Figure A.2 can be expressed and solved to illustrate the basic control structure embodied in the backward chaining approach. Provided some simplifying assumptions are made, the family tree in Figure A.2 can be expressed using a very few Prolog predicates:

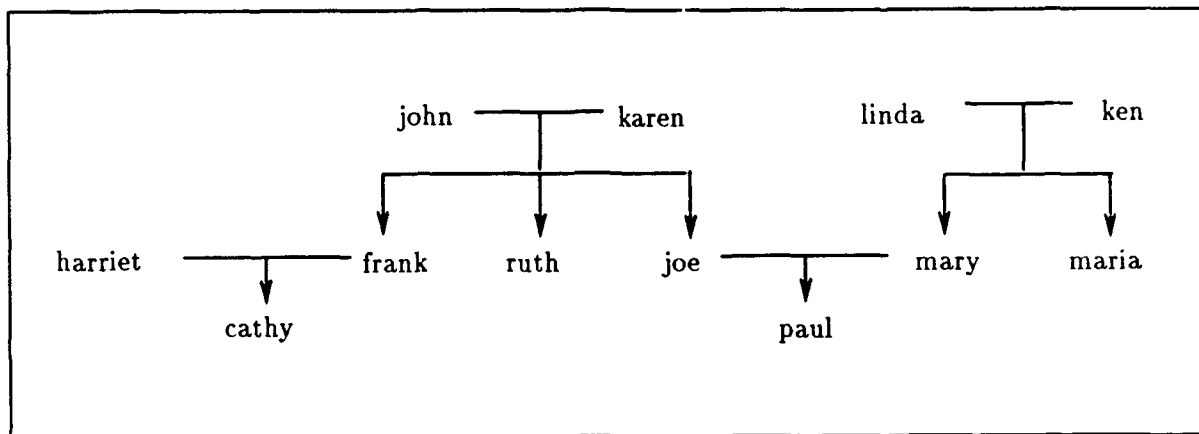


Figure A.2. Family Tree for the Example Prolog Problem

“Parent” Facts:

parent(john,frank).
 parent(john,ruth).
 parent(john,joe).
 parent(karen,frank).
 parent(karen,ruth).
 parent(karen,joe).
 parent(harriet,cathy).
 parent(frank,cathy).
 parent(joe,paul).
 parent(mary,paul).
 parent(linda,maria)
 parent(ken,maria)
 parent(linda,mary)
 parent(ken,mary)

“Sex” Facts:

male(john).
 female(karen).
 female(harriet).
 male(frank).
 female(cathy).
 female(ruth).
 male(joe).
 female(mary).
 male(paul).
 female(linda)
 female(maria)
 male(ken)

This definition of the family tree assumes that every married couple has at least one child and that every couple with a child is married. Given this problem statement, several basic

relationships about family members can be easily written as logical predicates:

$$\begin{aligned}\text{parent}(X,Y) \wedge \text{male}(Y) &\rightarrow \text{son}(Y,X). \\ \text{parent}(X,Y) \wedge \text{parent}(Y,Z) &\rightarrow \text{grandparent}(X,Z).\end{aligned}$$

The first predicate states that Y is a son of X if Y is a male and X is a parent of Y. The second predicate states that if X is a parent of Y and Y is a parent of Z, then X is a grandparent of Z. Using Prolog, these two logical relations can be written as:

$$\begin{aligned}\text{son}(Y,X) &:- \text{parent}(X,Y), \text{male}(Y). \\ \text{grandparent}(X,Z) &:- \text{parent}(X,Y), \text{parent}(Y,Z).\end{aligned}$$

The Prolog logical relations are all written in the form of one “head” predicate on the left hand side of the implication operator ($:-$) and a conjunction of predicates on the right hand side; this restricted form of logical relationship is known as a *Horn Clause*. As seen later, Prolog also allows right hand sides that are disjunctions as well, but these clauses with right hand side disjunctions are not Horn clauses in the strict sense of the word.

The example problem derived from the family description concentrates on defining, and solving for the relation “ $\text{aunt}(X,Y)$ ”. This example problem assumes that a number of other family relations (sister, brother and wife) have already been defined and can now be used in defining the “ $\text{aunt}(X,Y)$ ” relation. In the “real world”, there are two different ways for a person X to be the aunt of some person Y: First, the person X can be the sister of one of the parents of the person Y, a relation that can be written as:

$$\text{aunt}(X,Y) :- \text{parent}(A,Y), \text{sister}(X,A).$$

Another way a person X can be an aunt of a person Y is that person X is the wife of a brother of a parent of person Y, a relation that can be written as:

```
aunt(X,Y) :- parent(A,Y), brother(A,B), wife(X,B).
```

Separately written Prolog rules with the same “head” are implicitly separated by a logical “or” operator, therefore it is possible to specify the aunt relation using one rule, if it is written as:

```
aunt(X,Y) :- parent(A,Y), brother(A,B), wife(X,B)
            ;
            parent(A,Y), sister(X,A).
```

Once the “aunt” relation has been defined and the other facts and relations with respect to the family program are provided, it is possible to solve some simple problems using Prolog. Figure A.3 provides the script from a short session a Prolog interpreter. Using the Prolog *consult* facility, all of the basic facts and relations (including the “aunt” relation defined in the previous paragraph) are loaded into the Prolog system. From the Prolog prompt (“?-”), it is now possible to make queries about the family that Prolog can use its “expert” knowledge to solve. From the family tree, it is obvious that “maria” is an aunt of “paul” so the first query asks if this relation is true:

```
aunt(maria,paul)
```

Prolog returns a “yes” answer to confirm that this relation is indeed true. As discussed in Chapter 2, one of the most powerful facilities in production systems is pattern matching. The analagous pattern matching facilities of Prolog are revealed in the second query which asks: Who are all the aunts of “paul”?:

```

+-----+
| MS-DOS Prolog-1           Version 2.2 |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd. |
| Oxford U.K. |
+-----+

```

```
?- consult('appendix').
```

```
yes
```

```
?- aunt(maria,paul).
```

```
yes
```

```
?- aunt(X,paul).
```

```

X = ruth
More (y/n)? y
X = maria
More (y/n)? y
X = harriet
More (y/n)? y
no

```

```
?- aunt(X,Y).
```

```

X = ruth
Y = cathy
More (y/n)? y
X = ruth
Y = paul
More (y/n)? y
X = maria
Y = paul
More (y/n)? y
X = mary
Y = cathy
More (y/n)? y
X = harriet
Y = paul
More (y/n)? y
no

```

```
?- halt.
```

Figure A.3. A Short Session With A Prolog Interpreter

aunt(X,paul)

Given a query with one or more variables, the Prolog interpreter systematically attempts to match the variable with specific instantiations that will cause the relation to become true. For the example family tree, the Prolog interpreter returns the following valid answers for the last query:

X = maria X = ruth X = harriet

Even though this example is very simplistic, programmers not familiar with Prolog can gain some appreciation for how this language differs considerably from most procedural programming languages. Although the actions of the Prolog interpreter may appear very complex, it is actually based on a conceptually simple algorithm.

A.3 Resolution and Refutation in Prolog

Prolog combines automated predicate resolution and a common logical proof method, proof by refutation, to solve logically defined problems (54:199). This section discusses the concept of resolution and then provides an example of how Prolog combines resolution and proof by refutation¹ to solve a portion of the previous "aunt" relation.

The resolution rule discovered by J. A. Robinson in 1965 forms the basis for how Prolog derives answers from logically stated problems. Using the general resolution method, the unsatisfiability of any well-formed formula (in clause form) can be proven (55:140). In very simple terms, the resolution method states that given two logical clauses, e.g.,

$$L_1 \vee L_2 \vee Q \text{ and } L_3 \vee \neg Q \vee L_4 \vee L_5$$

¹It is assumed that the reader understands proof by refutation

	<u>Logical Clause</u>	<u>How Derived</u>
1.	$\neg \text{parent}(Z,Y) \vee \neg \text{sister}(X,Z) \vee \text{aunt}(X,Y)$	given
2.	$\text{parent}(\text{joe},\text{paul})$	given
3.	$\text{parent}(\text{mary},\text{paul})$	given
4.	$\text{sister}(\text{ruth},\text{joe})$	given
5.	$\text{sister}(\text{maria},\text{mary})$	given
6.	$\neg \text{aunt}(X,\text{paul})$	refuted query
7.	$\neg \text{sister}(\text{joe},X) \vee \text{aunt}(X,\text{paul})$	1,2 : $Z = \text{joe } Y, = \text{paul}$
8.	$\neg \text{sister}(X,\text{mary}) \vee \text{aunt}(X,\text{paul})$	1,3 : $Z = \text{mary } Y, = \text{paul}$
9.	$\neg \text{parent}(\text{joe},Y) \vee \text{aunt}(\text{ruth},Y)$	1,4 : $X = \text{ruth } Z = \text{joe}$
10.	$\neg \text{parent}(\text{mary},Y) \vee \text{aunt}(\text{maria},Y)$	1,5 : $X = \text{maria } Z = \text{mary}$
11.	$\text{aunt}(\text{ruth},\text{paul})$	2,9 : $Y = \text{paul}$
12.	$\text{aunt}(\text{maria},\text{paul})$	3,10: $Y = \text{paul}$
13.	\square (solution)	6,11: $X = \text{ruth}$
14.	\square (solution)	6,12: $X = \text{maria}$
15.	No other resolutions possible	

Figure A.4. Using Resolution To Solve the Aunt Problem

in which the item are arbitrary literals (constants), an item is *opposed* if it appears directly in one clause and is negated in the other (in this case Q and $\neg Q$) and a resolvent clause can be formed using the following three steps (55:134):

1. Form a clause containing all of the items in both clauses
2. Delete the opposed pair of items
3. Remove all redundant (repeated) items

Using the resolution method on the above pair of clauses yields the following resolvent clause (55:135)

$$L_1 \vee L_2 \vee L_3 \vee L_4 \vee L_5$$

It should be noted that Robinson's method supports the resolution of clauses that are not composed entirely of literals as the above simplified example implies (55:136). For a more extensive and formal discussion of the resolution method, the reader is referred to (55:125-145).

Since the resolution method proves the unsatisfiability of a given formula, using proof by refutation to actually provide an automated theorem proving capability seems appropriate. In the case of this example problem, a subset of the "aunt" problem is solved (actually proven) using the resolution method and proof by refutation. This proof illustrates approximately the algorithmic approach used by the Prolog interpreter. The first five entries in the proof table in Table A.4 represent the first "aunt" relation (in conjunctive normal form) and a selected subset of the family data (some of it processed) that applies to that rule. The sixth entry is the refutation of the example query "who are the aunts of paul?". At this point, it is merely a matter of attempting to successfully apply resolution to all possible combinations of two clauses until no more resolutions can be successfully performed. The *how derived* column in Figure A.4 determines which two clauses were used to provide the resultant clause in that row. Whenever the empty clause (\square) is produced by the successful resolution of two single item clauses, a solution has been found. The variable bindings that allowed a successful resolution to occur then become the answer to the original query. Note that the two answers produced by the example problem represent a subset of the answers produced by the Prolog interpreter in the previous section.

A.4 Parallelism in Prolog Problem Solving

The simple "aunt" example problem which has been the subject of interest in the past two sections is also very useful in providing a rudimentary description of parallelism in the Prolog problem solving process. There are essentially two types of parallelism in Prolog (3:186):

- "OR-Parallelism, from trying to solve a goal in different ways."
- "AND-Parallelism, from trying to solve several sub-goals at once."

Figure A.5 provides a visual description of how parallelism can be applied to part of the "aunt" problem. There are two different ways for a person X to be the aunt of another person Y, therefore OR-Parallelism can be used to easily explore both of these possibilities in parallel. Each of the alternative "aunt" problem approaches also contain several sub-goals making the use of AND-parallelism a possibility as well. Although both approaches

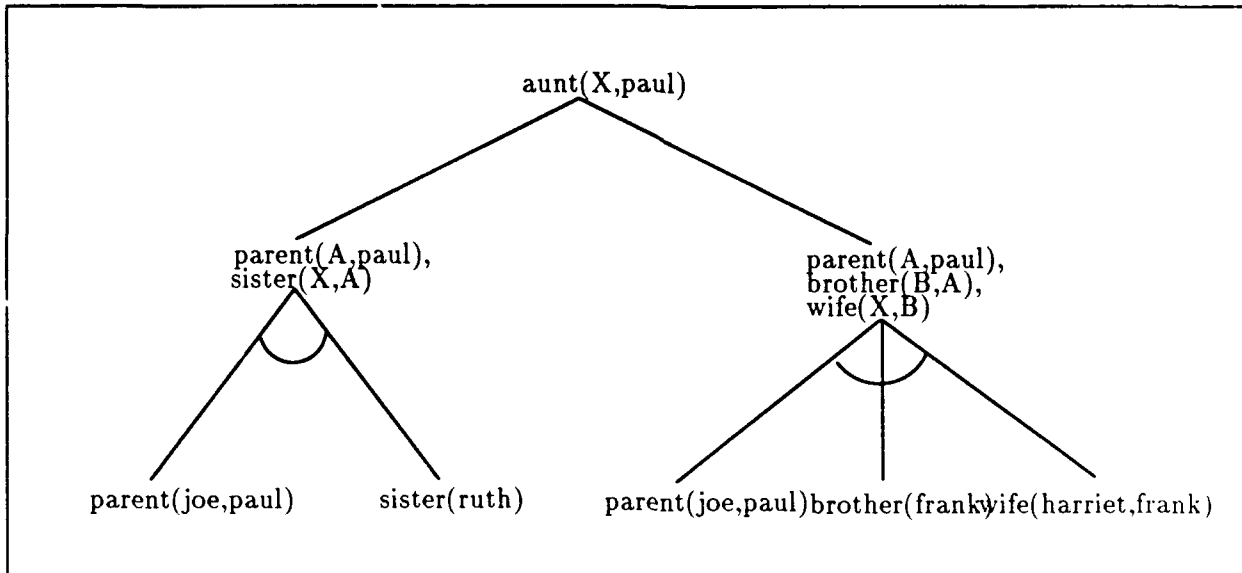


Figure A.5. A Representation For Part of The "Aunt" Problem

(OR and AND parallelism) look promising initially, it turns out that OR-parallelism is very easily implemented whereas effective AND-parallelism appears very difficult to achieve (3:186). Unfortunately, most problems appear to have large amounts of AND-parallelism, but very little OR-parallelism (3:187) (3:74).

Parallel matching of different rules in production systems is analagous to OR-parallelism in Prolog except that this type parallelism in Prolog is generally more constrained than in production systems. Prolog OR-Parallelism is relatively easy to achieve by replacing the normal Horn clauses used in sequential Prolog with guarded Horn clauses of the form:

$$A \leftarrow G_1, G_2, \dots, G_m \mid B_1, B_2, \dots, B_m$$

where the G_i elements are the guards and the B_i element are the problem subgoals (3:188). The use of guards on the Horn clause elements ensures that only one processor (that has "seized" the clause) can change the state of a given Horn clause, until that clause is "released" by the same processor. In the "aunt" problem, there could be two different guarded Horn clauses. Thus two processors could work independently toward solving this problem

with a maximum theoretical speed-up of 2. For the small "aunt" problem, an upper bound speed-up of 2 seems reasonable, however measurements on larger Prolog applications, specifically expert systems, reveal that OR-parallelism does not increase significantly with the size of the applications (23:78) (3:187).

Based on the limited amount of OR-parallelism available in most Prolog applications, the use of AND-parallelism appears mandatory in order to obtain significant speed-ups; however, the very nature of the resolution methodology (on which Prolog is based) appears to oppose AND-parallelism (3:188). The fundamental problem with AND-parallelism occurs because all the sub-goals in a purely conjunctive Horn clause share a common set of data and variables (3:187). The sharing of data and variables can lead to *variable binding conflicts* which may occur whenever two processors calculating different sub-goals of the same goal attempt to bind the same variable with different values. Consider the case for solving "aunt(X,Y)" using the approach represented in the left half of Figure A.5. If the variable bindings were not "coordinated" by the two processors solving the two different sub-goals, then one processor might return the satisfied clause "parent(joe,paul)" while the other processor could return the satisfied clause "sister(mary,maria)". Separately, both of the clauses are correct, but together they result in the incorrect conclusion "aunt(mary,paul)". Approaches to implementing AND-parallelism in Prolog which prevent binding conflicts are:

1. Compile-time techniques that automatically develop one or more alternative search graphs based on data dependency analysis (3:187).
2. Run-time techniques that attempt to perform clause resolution in a pipelined (or stream) manner (23:78).
3. Run-time techniques that rely on "weak" or "mock" resolution on sub-goals in parallel to reduce the search graph (23:78).

At this point in time, none of the above methods for implementing AND-parallelism have been tested extensively and thus no firm results are available (3:187) (23:78).

Appendix B. *A Discussion of the UNITY Approach*

B.1 Introduction

UNITY is more than just a high-level notation for expressing parallel program designs, it is a philosophy for developing correct and efficient parallel program designs and their subsequent implementation. Chapter 3, Section 2.2 covers much of the basic UNITY design process, but omits most of the details associated with actually constructing a UNITY “program” (design). This appendix presents the basic elements of the UNITY notation and demonstrates its use through some simple example programs. Although textual descriptions accompany all UNITY programs in the body of this document, the brief discussion of the UNITY notation in this appendix serves to make the notation used more understandable to readers not familiar with it. In addition to providing example programs, this appendix follows through on the initial design with a miniature UNITY program development cycle. After developing the initial program specification, the UNITY development cycle includes a proof of correctness, a mapping to a parallel architecture (which includes further refinements), and finally an implementation. Due to the time and space requirements, only the more important points of the implementation step are covered.

B.2 UNITY Concepts and Notation

Before delving into the syntax of the (UNITY) language, it is worthwhile to outline some of the most basic underlying principles on which UNITY is based:

1. **Non-Determinism** - The “main” part of most UNITY programs are built on a set of executable assignment statements. Using the concept of non-deterministic control (called the “fairness” rule) each statement is executed infinitely often (15:4).
2. **Absence of Control Flow** - UNITY programs do not specify control flow simply because it is not required for developing a high-level program design. Additionally, divorcing the control flow from program construction allows for greater flexibility in later mappings to different parallel computer architectures (15:5).

3. *Synchrony and Asynchrony* - The UNITY language supports both the synchronous and asynchronous assignment of variables within a UNITY program; this ability to support both synchronous and asynchronous events is the core of a unified theory of parallel programming (15:5).
4. *States and Assignments* - All UNITY programs (although they may not look it) are composed entirely of states and assignments. In this representation, the progression from state to state takes place through the parallel or sometimes sequential assignment of variables. The states and assignments model of UNITY programs is what supports the extraction of proofs from these programs (15:6).

Given the basic UNITY concepts, it is now important to understand the overall structure of a UNITY program; a UNITY program consists of the following sections:

1. **declare** - This section names all the variables to be used and their types.
2. **always** - This section defines how the value of certain variables manifest themselves as function of other variables.
3. **initially** - This section defines the initial values of selected variables before execution begins. Variables whose values are not initialized take on arbitrary initial values.
4. **assign** - This section consists of a set of statements which assign the values of variables sequentially or in parallel. The following are some common ways for expressing assignment statements: An assignment statement may contain a number of assignments to be executed in synchronously (all at once), referred to as assignment components:

$$a[1], a[2], a[3] := 0, 1, 2$$

or as a set of *assignment components* separated by || (the synchronous assignment component separator):

$$a[1] := 0 \parallel a[2] := 1 \parallel a[3] := 2$$

or as a quantified set of assignment components using the following notation:

$$\langle \parallel i : 1 < i \leq 3 :: a[i] := i - 1 \rangle$$

Note that all of the above synchronous assignments statements perform the same task and are therefore equal in this respect. Another related UNITY structure of use is the conditional synchronous assignment statement:

$$\begin{array}{ll} x, y := -1, -1 & \text{if } x \cdot y < 0 \sim \\ 0, 0 & \text{if } x \cdot y = 0 \sim \\ 1, 1 & \text{if } x \cdot y > 0 \end{array}$$

in which the values of x and y depend upon the synchronous evaluation and assignment above. Finally separate assignment statements (**not** assignment components), separated by a $[]$ symbol, can be executed *independently* (asynchronously) instead of “all at once” as with synchronous assignment components. An example of two assignment statements is (15:28):

$$\langle \parallel i, j : 0 \leq i \leq N \wedge 0 \leq j \leq N \wedge i \neq j :: U[i, j] := 0 \rangle$$

$$[] \langle \parallel 0 \leq i \leq N :: U[i, i] := 1 \rangle$$

When both of these assignment statements are executed, they produce an $N \times N$ matrix with only the diagonal elements set to 1 and all others 0.

The above notation should prove sufficient for the reader to interpret the example programs in this appendix. However, more information on the UNITY notation can be found in Chapter 2 of Chandy and Misra (15).

```

Program P4 {the odd-even transposition sort} P4
  assign
    <||i : 1 ≤ i < N ∧ even(i) :: y[i], y[i + 1] := sort2(y[i], y[i + 1])>
  []
    <||i : 1 ≤ i < N ∧ odd(i) :: y[i], y[i + 1] := sort2(y[i], y[i + 1])>
end {P4}

```

Figure B.1. UNITY Example Program 1 For Odd-Even Transposition Sort (15:438)

```

Program P5 {the odd-even transposition sort} P5
  declare  k : integer
  initially k = 0 [] y[0] = -∞ [] y[N + 1] = ∞
  assign
    <||i : 1 ≤ i < N ::
      y[i] := min(y[i], y[i + 1]) if (imod2 = kmod2) ~
      max(y[i], y[i - 1]) if (imod2 ≠ kmod2)
    >
    ||k := k + 1      if k < N
end {P5}

```

Figure B.2. UNITY Example Program 2 For Odd-Even Transposition Sort (15:439)

B.3 A UNITY Program Example

For the example problem, this appendix discusses the development of a parallel Odd-Even Transposition Sort adapted from Chandy and Misra. One of the most useful approaches for deriving an initial UNITY program (design) is to start with a high-level description of the sequential program and then “gather” all assignments that can be executed in parallel into multiple assignment statements (15:438). Figure B.1 and Figure B.2 illustrate two ways in which an odd-even transposition sort can be written using UNITY notation. Notice that program P4 contains two assignment statements (separated by the [] symbol) whereas program P5 contains only one more complex assignment statement. This appendix arbitrarily chooses to concentrate on program P5 for the remainder of the program development and implementation discussion.

Before attempting to prove the correctness of program P5, it is important to have a basic understanding of “how the program works”. First, the program declares an integer k which abstractly acts as a loop counter and an array of integers that holds the items to be sorted (in this case positive integers) in positions 1 through N . Initially, location 0 of this array is set to the value $-\infty$ and the $N + 1$ location of the array contains $+\infty$; the reason for arbitrarily assigning these particular values to those locations of the array is shown later. Finally, the single statement of the program describes the action of the program. The execution of this assignment statement can easily be described using the example in Figure B.3. Because this program contains only one assignment statement, the non-deterministic approach used by UNITY to select and execute statements does not affect the description of the execution sequence shown in this example as only one assignment statement can be selected. All components of the single assignment statement are executed synchronously until the value of k is equal to N . Once $k = N$, the single assignment statement is no longer eligible to execute and the state of the program no longer changes. The condition under which the program state becomes static is referred to as the *fix point* (FP). As seen in the following section, the FP is a crucial part of the program proof.

B.4 Proving UNITY Program

Proof of a UNITY program can be easy or difficult depending on how the program is written and the level of detail it contains. The temporal logic constructs of UNITY (*unless*, *ensures*, and *leads-to* (\mapsto)¹), in conjunction with traditional logic notation can be used to prove that the behavior of the UNITY program matches the initial program (design) requirements (15:40). Using the following notation for the UNITY state transition model:

$$\{p\} s \{q\}$$

where p is the program state before executing an assignment statement s and q as the program state after executing the assignment statement s (15:41). The program state are

¹Also affectionately known to UNITY designers as: *useless*, *unsure* and *leads-nowhere* (\leadsto |)

index	Array						k	Metric
	0	1	2	3	4	5		
value	$-\infty$	4	3	2	1	∞	0	6
	$-\infty$	3	4	1	2	∞	1	4
	$-\infty$	3	1	4	2	∞	2	3
	$-\infty$	1	3	2	4	∞	3	1
	$-\infty$	1	2	3	4	∞	4	0

Figure B.3. Two Example Parallel Odd-Even Transposition Sorts

usually written in the form of logical predicate relating to the state of different program variables such as: $k = N$ in the case of program P5. The UNITY temporal logic constructs can be written as:

- $p \text{ unless } q \equiv \langle \forall s : s \in F :: \{p \wedge \neg q\} s \{p \vee q\} \rangle$ – meaning that if a logical predicate p , describing the state of the program, is true at some point in time, then after executing an assignment statement s either (15:48):

1. the logical predicate q , describing the program state, never becomes true, and p is always true or
2. the logical predicate q , describing the program state, eventually becomes true, and p continues to be true at least until q becomes true.

- $p \text{ ensures } q \equiv (p \text{ unless } q \wedge \langle \exists s : s \in F :: \{p \wedge \neg q\} s \{q\} \rangle)$ – meaning that if a logical predicate p , describing the state of the program, becomes true, then p

remains true as long as the logical predicate q , describing the state of the program, is false and eventually q becomes true.

- *Leads-to* does not have a state transition definition like *unless* and *ensures*, but its behavior can be implied from the above definitions, specifically:

1. $(p \text{ ensures } q) \rightarrow (p \mapsto q)$
2. $(p \mapsto q) \wedge (q \mapsto r) \rightarrow (p \mapsto r)$

Essentially *leads-to* means that for $p \mapsto q$, once the state of a logical predicate p , describing the state of a program, becomes true, then the logical predicate q , describing the state of the program, will eventually become true; however, *leads-to* does not require that p remain true until q becomes true in order for q to eventually become true.

In general, a UNITY program proof proceeds in the following manner:

1. Define and prove the existence of an *Invariant* (something that is always true about the program design throughout its execution).
2. Define and prove the existence of a FP.
3. Define a progress property and show that the progress property holds (is true) until the FP is reached (i.e. *leads-to* the FP).

For the odd-even transposition sort, this approach is adjusted slightly; first, a good generalized strategy for defining the progress property (or metric) is the number of out-of-order pairs in the array; this metric can be formally defined as (15:431):

$$M = \langle +i, j : 0 < i < j \leq N \wedge y[i] > y[j] :: 1 \rangle$$

In program P5, it is always true that *unless* $M = 0$, the value of M decreases during each execution of the assignment statement. More formally stated, the Invariant can be defined as (15:431):

$$I = \langle \forall k : k > 0 :: M = k \mapsto M < k \rangle$$

The previous section describes the execution of the program and shows that the FP is reached when $k = N$ at which time it can be shown that $M = 0$ (15:438). In order to prove $true \mapsto FP$, it is sufficient to show the relation between M and FP as (15:431):

$$true \mapsto (M \leq 0)$$

by using induction on the description of the invariant. Using the metric description:

$$M \geq 0$$

until the time at which the FP is reached thus: $true \mapsto FP$.

B.5 Mapping to A Parallel Architecture

As noted in Chapter 3, one of the primary strengths of the UNITY approach is the ability to map programs to different parallel architectures while maintaining their correctness. This section considers the mapping of the odd-even transposition sort program into a distributed (memory) parallel architecture. In order to correctly perform this mapping, the three following steps must be performed (15:83):

- Allocate each statement in the program to a processor
- Allocate each variable in the program to a memory
- Specify the control flow for each processor

In addition to these three steps the following two constraints on this mapping must be met:

- All variables named on the left hand side of an assignment statement allocated to a processor must be available to that processor through the use of shared memory or communication channels (in the case of a distributed memory system).
- The flow of control for each processor follows the “fairness” rule (each statement is executed infinitely often)

By strictly following the UNITY notation, program P5 contains only one statement. Therefore it initially appears that only one processor can be used for this problem. However, by using UNITY's *Single Statement Schema* mapping, each of the assignment components in the one assignment statement can be assigned to a single processor. This particular *Single Statement Schema* mapping may appear to solve the problem of effectively dividing the problem among a number of processors, but it is not in itself correct without the constraints on communication channels explained in the following paragraph (15:88).

Distributed memory parallel architectures communicate through message passing over communication channels instead of through the use of shared memory. Therefore, the UNITY distributed memory program mapping must consider the proper use of communication channels in order to maintain program correctness. Specifically the two constraints that must be satisfied by distributed memory mapping are (15:83):

1. At most one variable is allocated to each communication channel and this variable is of type sequence.
2. A variable allocated to a channel is named in the assignment statements of exactly two processors. Assignment statements can modify a message sequence by either appending an item of data to the rear of the sequence (writing) or removing an item from the front of the sequence (reading). Thus, the message channel acts abstractly as a first-in-first-out queue.

When using the *Single Statement Schema* for mapping program P5, the approach remains correct as long as the interprocessor communication structure is linear (or capable of emulating a linear structure) and each communication channel holds only one message (15:441). When using the distributed memory architecture mapping described above, the refined UNITY program in Figure B.4 results.

B.6 Implementing Odd-Even.c on a Parallel Computer

Following the initial design, proof of correctness, architectural mapping and further iterative development, the UNITY program is ready for implementation on a selected parallel computer using a standard programming language. In the case of this example

```

Program P5 {the odd-even transposition sort} P5-Distributed-Memory
  declare  k : integer {Assume k is local to each processor}
  initially k = 0 [] channel[0] =  $-\infty$  [] channel[N + 1] =  $\infty$ 
  assign
    {For each processor i independently, perform the following statement:}
    ([i : 1 ≤ i ≤ N ::
      channel[i] := min(y[i], channel[i + 1]) if (imod2 = kmod2) ~
      max(y[i], channel[i - 1]) if (imod2 ≠ kmod2)
    )
    || k := k + 1          if k < N
    )
end {P5-Distributed-Memory}

```

Figure B.4. Distributed-Memory UNITY Program For Odd-Even Transposition Sort

development process, the target distributed memory parallel computer is an Intel iPSC/2 and the implementation language is a parallel extension of the C programming language modified specifically for use with the iPSC/2. Given the in-line comments in the program source code below, the reader should be able to follow the loose translation of the UNITY program P5-Distributed-Memory to the its “realization” in Odd_Even.c. It should be stressed that due to the communication overhead associated with this simple sorting implementation, program performance is likely to be very poor (worse than for a serial implementation). On the other hand, this program demonstrates that correctness can be maintained from the basic design through to the final implementation. Additionally, the basic design of this program can now be used as a “template” to design parallel odd-even transposition sorts that are capable of significantly more flexibility and better performance on a parallel distributed memory computer architecture.

```

/*****/
/*      Odd-Even Transposition Sort UNITY Code Translation      */
/*                                                                 */
/* Name: Odd_Even.c                      Author: Capt George Sawyer */
/*****/

#include <stdio.h>

```

```

#define CHANNEL      10
#define INFINITY     9999
#define NEG_INFINITY -9999

main() {

    int i;          /* holds the value of the processor used */
    int k;          /* holds the local value of the increment variable */
    int local;      /* holds this processors part of the array to sort */
    int remote;     /* holds the array value from another processor */
    int N = 4;      /* holds the number of elements in the array */

    int min();      /* declaration for the minimum of two vars function */
    int max();      /* declaration for the maximum of two vars function */

    i = mynode();
    local = N - i; /* initializes distributed array to a reverse order */

    /* Following loop continues to execute independently on each pro- */
    /* cesssor, until the value of k reaches the value of N (fix point) */

    for (k = 0; k < N; k++) {

        /* Case of the first "if" statement is true */

        if ((i % 2) == (k % 2)) {

            /* if the processor value is the upper limit: */

            if (i >= (N - 1))
                local = min(local, INFINITY);

            /* otherwise, proceed normally: */

            else {

                /* using send-wait and receive-wait ensures only */
                /* one message is in the channel at a time */

                csend(CHANNEL, &local, sizeof(local), (i + 1), 1);
                crecv(CHANNEL, &remote, sizeof(remote));
                local = min(local, remote);

            } /* end if */

        }
    }
}

```

```

    }    /* end if */

    /* Otherwise assume that the second case is true */

    else {

        /* if the processor value is the lower limit: */

        if (i == 0)
            local = max(local, NEG_INFINITY);

        /* otherwise, proceed normally: */

        else {

            /* using send-wait and receive-wait ensures only */
            /* one message is in the channel at a time      */

            csend(CHANNEL, &local, sizeof(local), (i - 1), 1);
            crecv(CHANNEL, &remote, sizeof(remote));
            local = max(local, remote);

        }    /* end if */
    }    /* end if */
}    /* end for loop */

/* each processor prints out "its" final value of the array */

printf("The local value for processor %d is: %d \n", i, local);

}    /* end main program */

/* function that returns the minimum value of two passed integers */

int min(x, y)
    int x,y;
{
    if (x <= y)
        return x;
    else
        return y;
}

/* function that returns the maximum value of two passed integers */

```

```
int max(x, y)
    int x,y;
{
    if (x >= y)
        return x;
    else
        return y;
}
```

```
/***/
```

Appendix C. *Detailed Discussion of Search Algorithms*

C.1 *Introduction*

In sections of Chapters 4 and 5 of this document, the concept of rule-to-processor partitioning for achieving maximum production parallelism is discussed, but not pursued in detail. This topic is one of significant importance to current WRDC expert system projects such as the Robotic Air Vehicle (RAV) and Pilots Associate (PA) programs. This appendix discusses, in detail, the techniques used by the STEPPS system to derive a rule-to-processor partitioning for evaluation of production parallelism. The methodologies used are based on the static characteristics of a production system application and do not consider the run-time characteristics of that system during the partitioning process. In the initial implementation, an A* search technique, which guarantees an optimal solution, was designed and implemented. Based on initial STEPPS system testing, the A* algorithm's $O(2^N)$ complexity results in unacceptably slow system performance for applications larger than 10 to 15 rules. Genetic algorithms do not guarantee optimal solutions; however they have been shown to produce acceptable quality results in polynomial time (33). For a set of small production systems applications, the genetic algorithm based search's best result was always within 10% of the optimum result found by the A* algorithm.

C.2 *A* Search Analysis and Implementation*

C.2.1 A Search Background* The A* search algorithm is an admissible search technique that dominates all other algorithms for searching "or" graph type problems. At the "heart" of the A* algorithm is the additive function f^* , which is the sum of two functions $g(n)$ and $h(n)$ where (63:75):

- $g(n)$ = cost of the path from the current state s to the next state n .
- $h(n)$ = an estimate of the minimum cost required to reach a goal state from state n .

Given the above description of the A* cost functions, the complete A* algorithm can be expressed in pseudo-code terms as (63:64):

1. Put the start state s onto a list of states OPEN.
2. If OPEN is empty, exit with failure.
3. Remove from OPEN, and place on another list CLOSED any state n for which f is a minimum.
4. If n is a goal node, exit successfully and determine the path to the solution by tracing back the path of state from n to s .
5. Otherwise expand the node n , generating all of its possible successors. For every successor n' of n :
 - (a) If n' is not a state currently on OPEN or CLOSED, calculate $f(n') = g(n') + h(n')$
 - (b) If n' is already on OPEN or CLOSED, update its state to reflect the lowest cost from s to n' .
 - (c) If n' was found on CLOSED, remove it from CLOSED and put it on OPEN.
6. Go to step 2.

The rule-to-processor partitioning problem is originally cast in the form of an A* search; however, the search tree to find an optimal solution can be generated using only one unique path to each state n . The state uniqueness of the rule-to-processor partitioning problem allows the A* search to assume the identity of a *branch and bound* type problem which significantly simplifies the above algorithm. In *branch and bound*, whenever a state can no longer lead to an optimal solution it is merely discarded instead of being put on a CLOSED list. Additionally, since each state has a unique path back to the start and this cost can be maintained by the state representation, there is no need to update the “f-cost” of states already on the OPEN or CLOSED lists.

C.2.2 The Optimal Partitioning Algorithm Implementation Chapter 3 of Brassard and Bratley's book *Algorithmics, Theory and Practice*, defines a search in terms of the following items (10):

- A list or set of available *candidates*.
- A list or set of *candidates* that have already been used.
- A function that checks whether a particular candidate *provides a solution*.
- A function that checks to whether a given candidate is *feasible*, in other words, if it is possible for a given candidate to reach a solution.
- A *selection function* to determine at any given point in the search space, what the most promising next candidate is.
- An *objective function* that gives the value of a solution; this is the function we are trying to optimize.

Using the semantic representation of the problem and the basic requirements of the search process, it is now possible to decompose the search process into its separate functions and then recombine them in the final search definition. The original problem representation defines the final solution as the matrix \mathcal{P} , where $\mathcal{P}_{x,y}$ represents whether or not rule y is allocated to processor x . Representing the current state of the search by which rules are currently allocated to which processor allows for direct and uncomplicated calculation of the span-of-effect associated with that allocation. Based upon this representation of the state, the related search functions can be defined as follows:

- The list of available *candidates*, also commonly known as the *OPEN* list, can be implemented as a priority queue of available states. Each state represents a number of possible rule-to-processor assignments having an associated span-of-effect cost. The higher the span-of-effect of a given state, the more promising it is and the closer to the front of the priority queue it is.
- In this search process, it is not necessary to keep a list of previously examined states, also commonly known as the *CLOSED* list, because these states are expanded into new states that are either put back into the priority queue or discarded because they cannot lead to an optimal solution.

- A state is a *possible solution* if all production rules are assigned to one and only one processor. Mathematically this can be expressed by:

$$\sum_{i=1}^n \mathcal{P}_{i,j} = 1 \quad \forall j \in 1..r.$$

- A rule to processor assignment is considered *feasible*, if the rule has not been assigned to a processor. It is possible to add additional constraints to the feasibility function in order to avoid redundant states in the search space. Restating the previous assumption concerning rule to processor assignment from Chapter 4, Section 5.3,

$$\mathcal{X} \subset \mathcal{Z}$$

$$Cost(\mathcal{X} \rightarrow P_1, \mathcal{Z} - \mathcal{X} \rightarrow P_2) = Cost(\mathcal{Z} - \mathcal{X} \rightarrow P_1, \mathcal{X} \rightarrow P_2),$$

it can be observed that the generation of numerous redundant states is possible. One of the possible reductions is to prevent rule r from being assigned to processor p whose number is less than that associated rule number. Mathematically, this procedure can be expressed as:

$$r \leftarrow \{\forall i \in n \mid \mathcal{P}_{ir} = 0\}$$

$$i \in n \quad \wedge \quad i \leq r.$$

- In all A* search procedures, the *selection* function is always:

$$f(state) = g(state) + h(state),$$

where $g()$ is the *objective* function (defined in the next item) evaluated to the current state and $h()$ is the *heuristic* function which returns the best cost estimate from the current state to the solution. In the case if this problem, we are attempting to obtain a maximum solution. Therefore, the heuristic function must overestimate the current state to solution cost in order to be admissible (63). The heuristic procedure used in this application is adapted from Dixit and Moldovan description of the allocation problem in production systems (22:26). For every unassigned rule, there is a

```

function A* ( $\mathcal{P}$  : Matrix of Boolean,  $\mathcal{S}$  : Matrix of Boolean):
( $OPEN$  is a list of candidate states)
 $OPEN \leftarrow \{\mathcal{P} \mid \mathcal{P}_{i,j} = 0 \ \forall i \in n, \forall j \in r\}$ 
While not  $OPEN = \phi$ 
   $t \leftarrow \{t \in OPEN \mid g(t) + h(t) \geq g(x) + h(x) \ \forall x \in OPEN\}$ 
   $r \leftarrow \{\forall i \in n \mid \mathcal{P}_{i,r} = 0\}$ 
  For  $i \in n \ \wedge \ i \leq r$ 
     $t' \leftarrow \{t \mid t_{i,r} = 1\}$ 
    If  $g(t') + h(t') \geq g(BEST) \ \wedge \ \sum_{j=1}^r t_{i,j} = 1 \ \forall i \in n$ 
       $BEST \leftarrow t'$ 
    If  $g(t') + h(t') \geq g(BEST) \ \wedge \ \text{not } \sum_{j=1}^r t_{i,j} = 1 \ \forall i \in n$ 
       $OPEN \leftarrow t'$ 

```

Figure C.1. The A* Search Rule-to-Processor Partitioning Algorithm

cost increase associated with assigning this rule to any processor. Summing up the maximum increases in cost associated with assigning an unassigned rule to a processor yields the maximum increase in span-of-effect. This process can be expressed mathematically as:

$$h(\mathcal{P}) = \sum_{i=1}^n MAX(g(\mathcal{P}) | (\mathcal{P}_{i,j} = 1) - g()) \quad \forall j \in r \ \wedge \ \sum_{i=1}^n \mathcal{P}_{i,j} = 0.$$

The terminology $g(\mathcal{P}) | (\mathcal{P}_{i,j} = 1)$ means the objective cost of the current state as defined by \mathcal{P} given that rule j is assigned to processor i .

- The *objective* function is the function we are trying to maximize. It was defined and explained earlier, but it is redefined here in terms of the state space search as:

$$g() = \sum_{i=1}^r \sum_{j=1}^n \sum_{k=1}^r \mathcal{P}_{i,j} \mathcal{P}_{k,j} \mathcal{S}_{k,i}.$$

Given the definition of the search components and the *branch and bound* simplification of the A* algorithm, the optimal search process is in Figure C.1.

C.3 Genetic Algorithm Based Search

C.3.1 Genetic Algorithms Background Genetic algorithms are search algorithms based on the mechanics of the natural selection process. The most basic premise of this

selection process simply states that the strong tend to adapt and survive and the weak tend to die out. As with the natural evolutionary process, probability plays a significant role, leading to the use of the “tends to” statements. By modeling the genetic algorithms after the process of natural selection, researchers hope to achieve a *robust* search technique which Goldberg describes as “a balance between efficiency and efficacy” (33:1). That is, the algorithm will produce “close” to optimum results in a “reasonable” amount of time instead of taking forever to find an optimum solution that may not exist or quickly finding a solution that is far from the optimum. The “balance concept” also states that the algorithm can be easily adapted to work on problems with diverse characteristics. Typically, the “traditional” problem solving approaches tend to perform well only on the problem for which they were specifically intended. Enumerative (exhaustive search) methods tend to work well across a large problem domain, but they are typically very inefficient. The *robust* scheme achieves a balance between these two extremes to achieve “good” performance across a wide range of problem types (33:6).

Genetic algorithm problem solving essentially concentrates on “evolving” a population of individuals over a number of generations using the genetic operators: selection, mutation and crossover. Each individual in the population typically represents a possible problem solution encoded as a sequential string of “1s” and “0s”. Using biological terms, each of these string positions corresponds to an allele. Each individual in the population possesses a fitness value that depends on how good the solution to the problem, represented by that individual’s “genetic make-up”, is. The fitness value is based on an evaluation of the individual’s alleles using a defined “payoff” function (33:26). During each successive generation, the three genetic operators are applied to the population as a whole:

- **Selection** is the process of choosing the most *fit* individuals from the current population that are later used to form the next generation; The *fitness* of a given individual is determined by a user defined objective function. The fitness of a given individual (with respect to the average fitness of the population) determines how many offspring this individual is expected to produce. The actual number of times a given individual is selected is dependent on not only the number of offspring it is expected to produce, but probability (chance) as well.

- **Mutation** randomly changes the value of given alleles in the population as a whole. Thus, the genetic “make up” of certain individuals in the population can change (usually just slightly) as a result of having one of its allele values randomly changed by mutation. This operation corresponds to the genetic process of mutation that occurs within all naturally adapting systems.
- **Crossover** exchanges alleles among a specified percentage of adjacent pairs in the population. Two *offspring* individuals are produced by the exchange of alleles between each two members of the population. These offspring individuals then replace their *parent* individuals in the population. This step, of course, mimics the natural process of procreation, but makes the assumption that non-identical (usually) twins are produced by each couple in the population to which the crossover operation is applied.

The intention of this process is to generate a more fit population of individuals during each successive generation until the average fitness of the population no longer increases.

C.3.1.1 An Example Problem Solution Like more traditional search algorithms, before the process of finding a solution can begin, we must first find a representation of the problem that is compatible with the search algorithm being used. In order to use genetic algorithms to solve a given problem, the problem must be in the form of a string of binary digits. Although not observed in this presentation, effective representation of the problem using a binary string is one of the most challenging aspects of using genetic algorithms (33:75). The state of the example problem can be easily represented using a sequence of five binary digits: x_4, x_3, x_2, x_1, x_0 (where $x_i = 1$ or 0). The value of any given string can be obtained using this formula:

$$f(x) = \left(\sum_{i=0}^4 (x_i)^i \right)^2$$

for example, the value of 00101 is 25. In addition to the values of $x_i = 1$ or 0 , another possible state for each digit is X which stands for a “don’t care” condition. If a certain bit position is a “don’t care”, we will make the simplifying assumption that it is not evaluated.

Genetic Algorithm Example						
String Number	Initial Population	x value	$f(x)$ x^2	p_{select} $f_i/\Sigma f$	Expected Count f_i/\bar{f}	Actual Count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4
Average			293	0.25	1.00	1
Max			576	0.49	1.97	2

Figure C.2. Genetic Algorithm Example (33:16)

Unlike traditional algorithms, genetic algorithms search from a population (set) of states instead of a single state. Typically, the initial population of points are chosen at random, but as discussed earlier, this random population can be “seeded” with promising individuals if some a priori knowledge about the problem is available. In the functional optimization example problem given by Goldberg, a coin is flipped 20 (4 individuals \times 5 alleles/individual) times to determine the initial population of four individuals in Figure C.2. One might argue that traditional search algorithms search from a set of states when they choose a state to expand. However, once a traditional search algorithm chooses a state, it turns all its “attention” on this selected states, temporarily excluding the other possible states from consideration. On the other hand, genetic algorithms consider the entire current population when choosing the subsequent population of states.

The first step in the genetic search process is to evaluate the population and use probabilistic techniques to **select** a set of promising individuals. Figure C.2 shows the $f(x)$ values associated with each of the individuals in the initial population. The p_{select} column of Figure C.2 shows the relative fitness of each individual with respect to the rest of the population. This fitness value, when multiplied by the size of the population, yields the number of offspring that each individual is expected to produce. The easiest way to visualize the actual selection process is through the use of the roulette wheel shown in Figure C.3. Each partition of the roulette wheel is proportional to the relative fitness

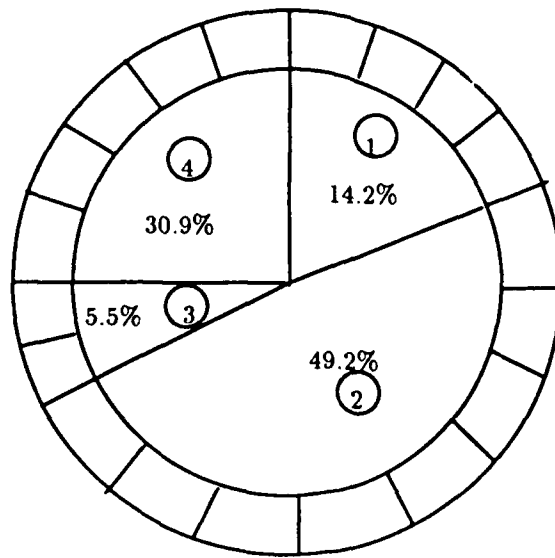


Figure C.3. Selection Roulette Wheel (33:11)

of each individual in the population. In Goldberg's example, spinning the wheel 4 times results in the individuals in the *Mating Pool* column of Figure C.4. At this point, the second genetic operator, mutation, can be applied.

In the **mutation** step, there is a random possibility that any one of x_i in any one of the individuals in the population will be altered. Assuming a mutation probability of 0.001 for the example of 4 individuals, each with 5 alleles, there will be $4 \cdot 5 \cdot 0.001 = 0.02$ mutations every generation (or 1 mutation every 50 generations) on the average. In this example, it is assumed that no mutations occur and that no members of the mating pool are affected. At this point, the third genetic operator, crossover can be applied.

In the **crossover** step, members of the mating pool are paired and mated (each resulting in two offspring) to form the new population. For each pair of individuals, a random *crossover-point* is chosen. Using the example in Figure C.5, it is fairly easy to see how the crossover step results in a new population of individuals. The hope is, that by mating the most fit individuals in the population, a population of even more fit individuals will be produced. In this case, the concept works because the average and maximum fitness of the new generation is greater than that of the previous population.

Genetic Algorithm Example							
String Number	Mating Pool After Reproduction (Cross Site Shown)		Mate	Crossover Site	New Population	x value	$f(x)$ x^2
1	0 1 1 0	1	2	4	0 1 1 0 0	12	144
2	1 1 0 0	0	1	4	1 1 0 0 1	25	625
3	1 1	0 0 0	4	2	1 1 0 1 1	27	729
4	1 0	0 1 1	3	2	1 0 0 0 0	16	256
Sum							1754
Average							439
Max							729

Figure C.4. Genetic Algorithm Example (cont) (33:17)

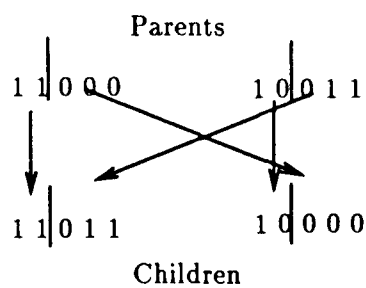


Figure C.5. Example Crossover Operation

After completing the crossover step, the evolution process is either terminated or returns to the selection phase where the new population of individuals becomes the focus. By continuing Goldberg's example, I was able to find the optimum result of 1, 1, 1, 1, 1 in just three generations.

C.3.1.2 Traditional vs Genetic Search From this example, it is fairly easy to see some of the basic differences between traditional and genetic search processes. Goldberg summarizes these differences as follows (33:22):

1. Direct manipulation of the problem state coding.
2. Search from a population, not a single point.
3. Search via sampling, essentially a blind search method.
4. Search using stochastic (probabilistic) not deterministic rules.

C.3.1.3 Genetic Algorithm Parameters Various parameters associated with the genetic algorithm such as population size, crossover rate and mutation rate can have a significant affect on the search process (33):

- *Population size*: Population size affects both global performance and a genetic algorithm's efficiency. Genetic algorithms with small populations usually perform poorly because the population provides an insufficient coverage of the problem space. A large population is more likely to be representative of the entire population and prevent premature convergence to a local instead of global solution.
- *Crossover rate*: The frequency with which the crossover operator is applied is controlled by the crossover rate. The higher the rate of crossover, the faster new structures are introduced into the population. If the crossover rate is too high, good performance structures are removed faster than selection can produce improvements. A low crossover rate on the other hand may stagnate the search.
- *Mutation rate*: To increase the variability of the population, mutation is used. A low level of mutation prevents a given position from freezing at a single value; a large mutation rate results in essentially random search.

It should be noted that these three parameters are only a few of the many parameters used by more complex genetic algorithm implementations. For a discussion of more advanced genetic algorithm concepts, the reader is referred to Goldberg (33), Chapters 3 and 4.

C.3.2 The Genetic Partitioning Algorithm Implementation Before using genetic algorithms as the basis for the rule-to-processor partitioning problem, a significant number of potential problem need to be resolved; among these problems are:

1. The binary string representation of the rule-to-processor partitioning for each individual in the population.
2. The design and implementation of the genetic algorithm itself, including selection, crossover, mutation and individual evaluation.
3. Selection of the genetic algorithm parameters that will allow the search to produce "good" results.

As noted in the previous section, the effective problem representation tends to be the most difficult part of using genetic algorithms. In comparison, the design and implementation of a genetic algorithm is relatively straightforward. Finally, selecting the proper genetic algorithm parameters is typically only a matter of sufficient experimentation.

C.3.2.1 Genetic Problem Representation Given the ILP problem representation in Chapter 4, the individual representation for the rule-to-processor partitioning problem is relatively straightforward. Given the 5 rule to 3 processor partitioning by the P matrix in Figure 4.11, the genetic string representation can be formed by simply concatenating the rows of the matrix together. Following the problem representation mapping stated above, the string equivalent produced is:

010001010001100

Needless to say, the evaluation equation must be altered to properly evaluate strings instead of matrices, but this is a relatively simple task. Although the above problem representation

mapping is very straightforward, it does have effects on the implementation of the genetic operators.

C.3.2.2 Genetic Algorithm Implementation The STEPPS genetic algorithm implementation is based on extension of David Goldberg's simple genetic algorithm (SGA) code in Chapter 3 of (33). Due to the problem representation stated above, some moderate changes to Goldberg's code were required for the crossover and mutation operators. If crossover and mutation were allowed to occur using the normal genetic operators, it is very possible that invalid problem solutions might result. For instance, mutating position 2 (from the left) of the string from the previous paragraph would produce the string:

000001010001100

in which rule 1 is not assigned to any processor. To ensure valid individuals are produced, crossover is allowed to occur only every N positions where N is the number of processors. Mutation is similarly altered so that only one mutation is performed on every N length group of "bits". If a given "group" of N "bits" is chosen for mutation, all values in that "group" become "0" and then one of the N values is randomly selected to take on the "1" value. Other modifications to the SGA involve the incorporation of Ada data abstraction/information hiding concepts and dynamic memory allocation for embedded use flexibility.

C.3.2.3 Genetic Algorithm Parameters Because this genetic algorithm development comprises only a small percentage of the total research, only a very limited amount of time was available to experiment with genetic algorithm parameters. Over a number of experiments, the results indicated that fairly small population of individuals (with respect to the number of rules), generally produced results equal to larger populations (up to the number of rules). To make the population dependent on the number of rules, but still small in comparison, and to make the population size even (for simplifying crossover) the figure of:

$$population_size := 2 \cdot \lceil \log_2 number_of_rules \rceil$$

was chosen. For a crossover value, a large value of 0.8 was chosen to allow new individuals into the population faster. Experiments showed that figure lower than 0.8 did not allow the new structures to be introduced quickly, and as a result, the evolution proceeded very slowly. A high mutation rate of 0.25 was also chosen because the search had a tendency to *stagnate* when the initial population did not contain any good individuals. Finally, the *elitist* strategy was added to the genetic search process to allow the "most fit" individual from a given generation to be propagated into the succeeding generation. Using this strategy ensured that once a close to optimal answer was discovered, it was not inadvertently discarded.

C.3.2.4 An Example GA Execution Following is a transcript of several generations of the genetic search process that attempts to optimize the 5 rule to 3 processor partitioning problem in Figure 4.9.

Generation 2 Statistics and Population:

***** The Population is: *****

The Best: 14 The Worst: 8 The Average: 1.15000000000000E+01

```
-----
0 0 1    0 0 1    0 0 1    1 0 0    0 0 1    fitness => 8
0 0 1    0 0 1    0 0 1    1 0 0    0 1 0    fitness => 14
0 0 1    0 0 1    0 0 1    1 0 0    1 0 0    fitness => 10
0 0 1    0 0 1    0 0 1    1 0 0    0 1 0    fitness => 14
-----
```

The parent selected is: 3

The parent selected is: 4

Crossing at position: 9

The parent selected is: 2

The parent selected is: 4

Crossing at position: 9

Generation 3 Statistics and Population:

***** The Population is: *****

The Best: 14 The Worst: 10 The Average: 1.25000000000000E+01

```
-----
0 0 1    0 0 1    0 1 0    1 0 0    0 1 0    fitness => 12
0 0 1    0 0 1    0 0 1    1 0 0    1 0 0    fitness => 10
0 0 1    0 0 1    0 0 1    1 0 0    0 1 0    fitness => 14
0 0 1    0 0 1    0 0 1    1 0 0    0 1 0    fitness => 14
-----
```

The parent selected is: 4
 The parent selected is: 3
 No Crossover!
 The parent selected is: 3
 The parent selected is: 2
 Crossing at position: 12
 Mutation Performed! The offset value is: 10
 Generation 4 Statistics and Population:
 ***** The Population is: *****
 The Best: 14 The Worst: 10 The Average: 1.300000000000000E+01

```

-----
0 0 1    0 0 1    0 0 1    1 0 0    0 1 0    fitness => 14
0 0 1    0 0 1    0 0 1    1 0 0    0 1 0    fitness => 14
0 0 1    0 0 1    0 0 1    1 0 0    1 0 0    fitness => 10
0 0 1    0 0 1    0 0 1    1 0 0    0 1 0    fitness => 14
-----
  
```

Appendix D. *STEPPS Users Guide*

D.1 Introduction

The Software Tool for Evaluating Parallelism in Production Systems (STEPPS) system, version 1.0 (31 October, 1990), is a rudimentary tool for statically analyzing the characteristics of a production system application in order to assess the level of parallelism present in that application. Due to the dynamic characteristics of production systems applications, STEPPS can only provide an “upper-bound” estimate on the level of parallelism using two different decomposition approaches:

- production parallelism
- node parallelism

Although a number of other decomposition approaches exist, analysis shows that even the upper-bound level of parallelism obtainable from these approaches cannot accurately be *determined*. The *underlying considerations and constraints associated with the STEPPS parallelism analysis approach* are explained in detail in Chapter 2 through 4 of this document. Chapter 5 of this document contains a detailed description of the STEPPS requirements, design and implementation. This guide provides sufficient information for the reader to successfully use STEPPS to analyze a given production system application, but it does not explain how to analyze the results or how these results were derived. In order to obtain an accurate interpretation of the STEPPS derived information, the user must read the information cited immediately above. In addition to providing basic information on how to run STEPPS, this guide also describes the limitations and cautions associated with the current implementation. Finally, this guide explains how to obtain the STEPPS system and compile it for use on any system with capable of supporting a validated Ada compiler and with sufficient memory to compile and execute the source code.

D.2 Using STEPPS

This section describes the inputs required by the STEPPS system and the outputs generated as a function of those inputs. An example STEPPS program execution is presented to clarify the input and output descriptions.

D.2.1 STEPPS Inputs The input to the STEPPS program consists of three separate items:

1. The STEPPS program command line options used to control STEPPS processing (required).
2. The source code file of the application to be analyzed (required).
3. Rule weighting information obtained by executing the application to be analyzed (optional).

This section of the STEPPS User's Guide describes each of these input items in detail and illustrates their use.

D.2.1.1 STEPPS Command Line The STEPPS interactive user interface is restricted to the command-line in a manner similar to many program language compilers. Because the command-line is the only direct link between the user and STEPPS, all processing must be specified at the time the program is invoked. Using UNIX parlance, the description of STEPPS is:

STEPPS(1)

NAME

steps - software tool for evaluating parallelism in production systems

SYNOPSIS

steps source-file [-w weight-file] [-r int] [-p int] [-l int] -cdmnsa

DESCRIPTION

STEPPS is a tool for analyzing the static characteristics of production system applications and assessing the level of parallelism inherent in that application from those system characteristics. Following are the required and optional arguments associated with the STEPPS program:

source-file - This is the file containing the source code for the production system application under investigation, this is the only argument required by STEPPS.

[-w weight-file] - This (weighting) flag and file-name argument are used to include rule weighting information into the STEPPS processing.

[-r int] - This (rules) flag and (positive) integer argument value are used to change the default number of rules expected by STEPPS. The default value is 36 rules; therefore, for a system with 40 rules the option "-r 40" should be included on the command-line.

[-p int] - This (pattern) flag and (positive) integer argument are used to change the default average number of LHS condition per rule expected by STEPPS. The default value is an average of 4 LHS conditions per rule; therefore, when processing a rule base with an average of 6 conditions per rule, the option "-r 6" should be included on the command-line.

[-l int] - This (limit) flag and (positive) integer argument are used to specify the maximum number of processors STEPPS is to partition the rule base over. The default value depends on the application under test and can be as large as the number of rules in that application.

[-c] - This (case) flag causes all "if" statements on the RHS of CLIPS to be assumed true. This means that the actions specified following all CLIPS RHS "if" statements will be considered in the parallelism analysis.

[-d] - This (dump) flag causes STEPPS to output all information computed during a given program "run". It activates all of the other print options discussed below:

[-b] - This (base) flag causes STEPPS to output the internal representation of all production rules read from the source file along with corresponding rule weighting information.

[-m] - This (matrix) flag causes STEPPS to output the internal matrix representations of the rule base information.

[-a] - This (allocation) flag causes the rule to processor assignments generated by STEPPS to be printed out.

BUGS

No explicit program bugs are currently known, however the system has significant limitations concerning the complexity of the production rules that can be successfully input to the system due to the use of static length

data structures. See the STEPPS User's Guide for detail. Additionally, source files not previously syntax checked by a production system interpreter may result in erratic or erroneous program behavior.

D.2.1.2 The Source File Input Application source files are the actual "programs" written in one of the production system languages accepted by STEPPS. For STEPPS 1.0, only applications written in CLIPS (version 4.0 through 4.3) and OPS-5 are accepted. Descriptions of these languages are contained in The CLIPS Reference Manual (18) and the OPS-5 User's Manual (27). STEPPS assumes that each application has been successfully processed by the respective interpreter thus ensuring that only rules with correct syntax serve as input. This is a vital assumption because STEPPS performs very little error checking with respect to the application source code. By eliminating much of the syntax error checking, STEPPS can accept either CLIPS or OPS-5 rules without the need to actually specify which language the application is written in! Not having to specify which language an application is written in makes STEPPS easier to use, but it also levies some very minor constraints on the use of the system. These constraints are outlined in a Section 3 of this guide.

D.2.1.3 The Weighting File Input The rule weighting file provides some very high-level run-time data that allows STEPPS to make more accurate judgments of the parallelism within a given application. In the absence of rule weighting information, STEPPS must assume that all rules are executed an equal number of times. In the case of many applications, this assumption is far from accurate and would have a negative affect on the accuracy of the STEPPS parallelism assessment. To alleviate this potential source of inaccuracy, STEPPS is able to read information regarding the number of times a given rule is executed to total number of rules executed by the system during an average "run". The weighting information is contained in an ASCII weighting file composed of rule-names on odd numbered lines and the floating point weights on the even numbered lines. The following subsection contains a sample rule weighting file.

Clearly, constructing a rule weighting file for a large application by hand would be an arduous undertaking. Therefore, STEPPS utilities are provided that allow easy extraction

and processing of rule weighting information. Using the STEPPS utilities, constructing the rule weighting file requires only a simple two-step process:

1. Capture the execution trace(s) of the application to be analyzed in a script file.
2. Process the script file using the UNIX *awk* utility and the *awk* programs shown in Figure D.1 or D.2.

Application execution traces are relatively easy to capture using OPS-5 and CLIPS production systems using the following five step procedure:

1. Prior to running the application under CLIPS or OPS-5 use the UNIX *script* facility to capture the program output.
2. Assert the proper watch level to enable the printing of rules as they are executed:
 - (a) CLIPS: Before asserting “(run)”, enter “(watch rules)” at the CLIPS prompt (18:61).
 - (b) OPS-5: Before asserting “(run)”, enter “(watch 1)” at the OPS-5 prompt (27:48).
3. When the program has completed, use “exit” to finish the script file.
4. If multiple “runs” are to be used in constructing the weighting file, merely concatenate the script files for those “runs” into a single file.
5. Use the appropriate *awk* program to process the script file:
 - (a) CLIPS: Given the “awkclips” program in Figure D.1, enter the following: “awk -f awkclips typescript > file.wgt” at the command-line.
 - (b) OPS-5: Given the “awkops5” program in Figure D.2, enter the following: “awk -f awkclips typescript > file.wgt” at the command-line.

After accomplishing the above actions, the weighting file for the application being analyzed will be in the file “file.wgt” ready for direct use by the STEPPS system.

```

$1 == "FIRE" {
    rule[substr($3, 1, (length($3) - 1))] = rule[substr($3, 1, (length($3) - 1))] + 1
    total = total + 1
}
END {
    for (name in rule)
        printf "%s\n%f\n", name, (rule[name]/total)
}

```

Figure D.1. The "awkclips" Rule Weight Processing Program

```

substr($2, 1, 4) == "fire" {
    rule[substr($2, 6, (length($2) - 7))] = rule[substr($2, 6, (length($2) - 7))] + 1
    total = total + 1
}
END {
    for (name in rule)
        printf "%s\n%f\n", name, (rule[name]/total)
}

```

Figure D.2. The "awkops5" Rule Weight Processing Program

D.2.2 STEPPS Program Outputs The STEPPS program outputs are mostly dependent on the command-line options chosen by the system user. The information that can or is produced by STEPPS includes the following:

1. Application-file-processing status information.
2. Program verification and debugging information.
3. Rule-to-processor partitioning assignments.
4. Available parallelism information.
5. Salient statistics of the rule base.

As rules are read from the input file, the names of the rules are printed out to the user. This rule-name printing is always enabled and cannot be disabled using command-line options. A significant amount of information that can be printed out by STEPPS has very little use except for program verification and/or debugging. This information includes the following:

1. STEPPS internal representation of the rules processed, including the dynamic weighting information for each rule processed.
2. The set of rule LHS patterns processed by STEPPS.
3. The A , B , C , and S , matrix representations of application rule base produced by STEPPS.

Because this verification/debugging information is typically not required by the system user, it is only printed out when specific command-line options are chosen. For applications containing a large number of rules, the rule-to-processor assignment results produce a large amount of output. Therefore, the printing of this information is disabled unless the appropriate command-line option is invoked. It should be noted that disabling the printing of rule-to-processor assignment information does not disable the printing of the production parallelism evaluation for each rule-to-processor assignment. Upper-bound information on production and node parallelism is always printed out as the primary function of STEPPS is to evaluate parallelism. Finally, STEPPS always prints out the

average rule base characteristics, however the printing of rule by rule characteristics is only enabled by command-line options.

D.3 STEPPS Cautions and Limitations

STEPPS is a research tool built on a very compressed schedule, therefore a number of cautions must be observed when using it to analyze production system applications. Basically, the STEPPS cautions and limitations fall into three categories: static data structure limitations, rule complexity limitations and language compatibility cautions. If the cautions and limitations specified in this section are observed, STEPPS possesses sufficient capability to analyze and evaluate the level of parallelism in large (several hundred rules) production systems applications.

D.3.1 Static Data Structures Limitations Static length data structures used in the STEPPS implementation result in the following limitations:

1. The names of the production rules must be less than or equal to 32 characters in length.
2. The length of any LHS condition, of any production rule, must be less than or equal to 128 characters. This restriction includes the length of "compound" LHS conditions composed of logical "or" conditions.
3. The length of any RHS action, of any production rule, is constrained in the same manner as the LHS conditions of production rules. Compound RHS actions such as "if-then-else" actions are also restricted to a length of 128 characters.

D.3.2 Rule Complexity Limitations Although STEPPS supports many complex expressions of CLIPS and OPS-5 rules, the following are limitations on the complexity of the production rules accepted:

1. STEPPS only supports one level of "or" condition nesting on the LHS of production of application production rules. Within any "or" condition, only one level of "and" condition nesting is accepted. For instance, an "or" condition could contain two

conditions, and either of these two conditions could be composed of a number of conditions combined by an explicit "and" (a "compound" condition) or just a number of single (non-compound) conditions. See pages 26 and 27 of the CLIPS Reference Manual (18) for more details.

2. STEPPS only supports production rules with a limited number of explicit LHS "or" conditions. Each condition within an inclusive LHS "or" condition actually requires the creation of a separate rule in order to maintain compatibility with the Rete algorithm. STEPPS is limited to 12 implicit rules combined within the body of a single production rule. For instance, a rule with two "or" compound-conditions (one with 2 simple conditions and the other with 4 simple conditions), would produce 8 different rules. See pages 26 and 27 of the CLIPS Reference Manual (18) for more details.
3. STEPPS only supports one level of "if-then-else" nesting on the RHS of production rules. Although CLIPS allows an arbitrarily deep level of "if-then-else" nesting, the use of this feature is not considered to be good production systems programming practice (18:56) and is not supported by STEPPS.
4. STEPPS does not support the use of the "while" construct that can be used on the RHS of CLIPS production rules.
5. STEPPS supports a maximum of 36 non-negated LHS conditions per production rule, but does not place any restrictions on the number of negated LHS conditions or RHS actions per rule. STEPPS also places a limitation of 40 different variables on the LHS of any given production rule.

D.3.3 Language Compatibility Cautions In order to simplify the use of the STEPPS system, the user does not have to specify which production system language (OPS-5 or CLIPS) the application under study is written in. This was done to reduce the size of the executable program and to make it more efficient. The assumptions used to implement this "transparent language acceptance" requires that some very simple precautions be taken in order to ensure the correct operation of the STEPPS system. The RHS actions names relating to the manipulation of working memory (WM) have different names in OPS-5

and CLIPS. In OPS-5, placing a new fact into WM is done with the "make" command and taking a fact out of WM is done with the "remove" command. The corresponding commands in the CLIPS language are "assert" and "retract". When using STEPPS to evaluate an OPS-5 application, care must be taken to ensure that all externally defined RHS functions using the CLIPS keywords "assert" and "retract" are removed. Similarly, when analyzing a CLIPS application, any externally defined RHS functions using the OPS-5 keywords "make" and "remove" must be discarded. The analysis of **most** applications will not be affected if these cautions are not observed. However, the user needs to be made aware of the possibility for incorrect results if the cautions outlined above are not considered.

D.4 Obtaining and Building STEPPS

For information on obtaining the STEPPS system source code, see Appendix E of this document. This appendix contains a complete list of the STEPPS source code files in correct compilation order. The source code was compiled and run using the Verdix Ada Development System (VADS), a validated Ada compiler. Thus, the source code should be completely compatible (the only exception is the command-line interface package) with other validated Ada compilers. The command-line interface package is a relatively small and simple piece of code which can be easily replaced by a system specific interface.

Appendix E. *STEPPS System Source Code*

E.1 Introduction

The names and contents of the Ada source files required for the STEPPS system, version 1.0 are listed below. Due to the space required for a listing of this source code, it is not included in this document. The source code files for the STEPPS system are maintained by the Department of Electrical and Computer Engineering at the Air Force Institute of Technology. The generic package listings, shown below in all capital letters, are licensed use software components and are therefore not available as part of the source code package.

E.2 File Names and Contents

Support Sub-System Packages:

VSTORAGES.a	Generic Sequential Storage Package Specification
BSTORAGES.a	Generic Sequential Storage Package Body
VINPUTFIL.a	Generic Input Filter Utility Specification
BINPUTFIL.a	Generic Input Filter Utility Body
VCHARUTIL.a	Character Utilities Package Specification
BCHARUTIL.a	Character Utilities Package Body
VSTRINGTL.a	Generic Unbounded String Package Specification
BSTRINGTL.a	Generic Unbounded String Package Body
VSTRINGTL.a	Generic Unbounded String Utilities Specification
BSTRINGTL.a	Generic Unbounded String Utilities Body
VLISTSUM.a	Generic Unbounded List Package Specification
BLISTSUM.a	Generic Unbounded List Package Body
VQUEPNSMN.a	Generic Unbounded Priority Queue Specification
BQUEPNSMN.a	Generic Unbounded Priority Queue Body
Matrix_Pkg.a	Two-Dimension Integer Matrix Package (Body & Spec)
Random.a	Low-Level Random Number Generation Package

System Parser Sub-System Packages:

LL_Parser_Spec.a	Low-Level String to List Package Utilities Spec .
LL_Parser_Body.a	Low-Level String to List Package Utilities Body
CE_Pkg_Spec.a	Condition Element Manipulation Package Spec.
CE_Pkg_Body.a	Condition Element Manipulation Package Body
Rule_Pkg_Spec.a	System Rule Manager Package Specification
Rule_Pkg_Body.a	System Rule Manager Package Body

Matrix Representation Sub-System Packages:

Random_Pkg.a	High-Level Random Number Generation Package
Genetic_Pkg.a	Genetic Algorithm Based Search Package (Body & Spec)
State_Package.a	A* Search State Representation Package (Body & Spec)
A_Star_Search.a	A* Algorithm Based Search Package (Body & Spec)
Matrix_Maker.a	Rule Base Matrix Representation Package (Body & Spec)

Evaluation Sub-System Packages:

Eval_Package.a	Parallelism Evaluation Package (Body and Spec)
----------------	--

Main

Commandi.a	Command-Line Interface Package
Stepps_Main.a	Main STEPPS Program Package

Appendix F. *Multi-CLIPS System Source Code*

F.1 Introduction

Multi-CLIPS is a modification of CLIPS version 4.3, a listing of the modified source code files is shown below:

```
engine.h  
engine.c  
factmng.c  
syssecnd.c
```

Because the current Multi-CLIPS system is an initial design and not particularly robust, it is not currently available to other than AFIT researchers. Future versions of Multi-CLIPS will be available to licensed users of CLIPS version 4.3 and above. CLIPS version 4.3, is available through the Computer Software Management and Information Center (COSMIC), the distribution point for NASA software. Further information can be obtained from

```
COSMIC  
382 E. Broad St.  
Athens, GA 30602  
(404) 542-3265
```

Due to the space required for a listing of the Multi-CLIPS source code, it is not included in this document.

Vita

Captain George A. Sawyer was born on 20 October 1962 in Peterborough, New Hampshire. He graduated from Fall Mountain Regional High School, Langdon New Hampshire in 1981. He attended the University of New Hampshire and on 25 May 1985, received the degree of Bachelor of Science in Electrical Engineering. Upon graduation, he received a reserve commission in the United States Air Force as a distinguished graduate of the Reserve Officer Training Corps program. He then served as a Flight Test Engineer and Director for the 513th Test Squadron, Strategic Air Command, Offutt AFB, NE, until entering the School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, in May 1989.

Permanent address: High Street
Walpole N.H. 03608

Bibliography

1. Abdelrahman, T. S. and T. N. Mudge. "Parallel Branch-and-Bound Algorithms on Hypercube Multiprocessors." In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 1492-1499, The Association for Computing Machinery, 1988.
2. Agha, Gul. *Actors; A Model of Concurrent Computation in Distributed Systems..* Cambridge MA: MIT Press, 1986.
3. Almasi, George S. and Allan Gottlieb. *Highly Parallel Computing.* Redwood City, CA: Benjamin Cummings Publishing Company, 1989.
4. Asberry, Raymond and others. *Parallel Programming Primer.* Technical Report, Intel Corporation, 1990. Beaverton OR.
5. Automated Reasoning Tool (ART) Reference Manual. ART Version 4.0; Inference Corporation Publication. April 1988.
6. Bab, Robert G. and others. *Programming Parallel Processors.* Reading, MA: Addison-Wesley Publishing Co., 1988.
7. Bechtel, Robert and Micheal C. Rowe. "Parallel Inference Performance Prediction." In *Proceedings of the National Aeronautics and Electronics Conference*, Volume 1, pages 189-199, IEEE, 1990.
8. Booch, Grady. *Software Components with Ada; Structures, Tools and Subsystems..* Menlo Park, California: Benjamin/Cumming Publishing Co., 1987.
9. Booch, Grady. *Object Oriented Design with Applications..* Menlo Park, California: Benjamin/Cumming Publishing Co., 1991.
10. Brassard, Giles and Paul Bratley. *Algorithmics, Theory and Practice..* Englewood Cliffs, NJ: Prentice-Hall, 1988.
11. Bratko, Ivan. *Prolog Programming For Artificial Intelligence..* Reading, Massachusetts: Addison-Wesley Publishing Co., 1986.
12. Brooks, Frederick P. "No Silver Bullet, The Essence and Accidents of Software Engineering," *IEEE Computer*, 28:10-19 (April 1987).
13. Brown, Frank M. EENG592 Class Handout: Problems in Prolog Programming # 2, Air Force Institute of Technology, Fall Quarter, 1989.
14. Brownston, L., R. Farrell, E. Kant and N. Martin. *Programming Expert Systems in OPS-5.* Reading, MA: Addison-Wesley Publishing Co., 1985.
15. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design: A Foundation.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1988.
16. Christofides, Nicos. *Graph Theory: An Algorithmic Approach.* London, England UK: Academic Press, 1975.

17. CLIPS Architecture Manual. Software Architecture of Version 4.3 of CLIPS; Artificial Intelligence Section, Lyndon B. Johnson Space Center. June 1989.
18. CLIPS Reference Manual. Version 4.3 of CLIPS; Artificial Intelligence Section, Lyndon B. Johnson Space Center. June 1989.
19. Coffman, Edward G. Jr. and Peter J. Denning. *Operating Systems Theory*. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1973.
20. Cohen, Norman H. *Ada As A Second Language*. New York, New York: McGraw-Hill Book Company, 1986.
21. Dally, William J. "Fine-Grain Message-Passing Concurrent Computers." In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 2-12, The Association for Computing Machinery, 1988.
22. Dixit, V. V. and D. I. Moldovan. "The Allocation Problem in Parallel Production Systems," *Journal of Parallel and Distributed Computing*, 8(1):20-29 (January 1990).
23. Douglass, Robert J. "A Qualitative Assessment of Parallelism in Expert Systems," *IEEE Software*, 2:70-81 (May 1985).
24. Duncan Ralph. "A Survey of Parallel Computer Architectures," *IEEE Computer*, 23:5-16 (February 1990).
25. Fanning, 1Lt Jesse. AFWAL Robotic Air Vehicle (RAV) Project Member. Telephone interview. Air Force Wright Aeronautics Laboratory, Wright-Patterson AFB, OH. 22 November 1989.
26. Flynn, M. J. "Very High-Speed Computing Systems." In *Proceedings of the IEEE*, pages 1901-1909, IEEE, December 1966.
27. Forgy, Charles, "The OPS-5 Users Manual." Department of Computer Science, Carnegie-Mellon University, Pittsburg, Pennsylvania. July 1981.
28. Forgy, Charles L. The OPS-83 Report and Users Manual, Production System Technologies, Pittsburgh, PA. May 1987.
29. Forgy, Charles L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, 19:17-37 (September 1982).
30. Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.
31. Fox, Geoffrey C. "What Have We Learnt from Using Real Parallel Machines to Solve Real Problems." In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 897-955, ACM, January 1988.
32. Giarantano, Joseph C. and Gary Riley. *Expert Systems: Principles and Programming*. Boston MA: PWS-Kent Publishing Co., 1989.
33. Goldberg, David E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, Massachussets: Addison-Wesley Publishing Company Inc., 1989.
34. Graham, R. L. "Bounds on Multiprocessing Timing Anomalies," *SIAM Journal on Applied Mathematics*, 17(2):416-429 (March 1969).

35. Gupta, Anoop. *Parallelism in Production Systems: The Sources and the Expected Speedup*. Technical Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1984. Contract F33615-81-K-1539.
36. Gupta, Anoop. *Parallelism in Production Systems*. MS thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March 1986.
37. Gupta, Anoop and Charles L. Forgy. *Measurements on Production Systems*. Technical Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1983. Contract F33615-81-K-1539.
38. Gupta, Anoop and Charles L. Forgy. "Static and Run-Time Characteristics of OPS5 Production Systems," *Journal of Parallel and Distributed Computing*, 8(1):20-29 (January 1990).
39. Gupta, Anoop and Milind Tambe. "Suitability of Message Passing Computers for Implementing Production Systems." In *Proceedings of the National Conference on Artificial Intelligence*, pages 687-692, August 1988.
40. Gupta, Anoop, and others. "Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis," *International Journal of Parallel Programming*, 17(2):95-124 (April 1988).
41. Harding, Capt William A. *Hypercube Expert System Shell - Applying Production Parallelism*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1989.
42. Harding, William A., George A. Sawyer and Gary L Lamont. "Hypercube Expert System Shell - Applying Production Parallelism." In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 18-27, ACM, April 1990.
43. Harvey, Wilson and others. *Measuring Effectiveness of Task-Level Parallelism for High-Level Vision*. Technical Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1989. Contract F33615-87-C-1499.
44. Hillyer, Bruce K. and David Elliot Shaw. "Execution of OPS5 on a Massively Parallel Machine," *Journal of Parallel and Distributed Computing*, 3:236-268 (June 1986).
45. Hoare C. A. *Communicating Sequential Processes*. Englewood Cliffs NJ: Prentice-Hall International, 1985.
46. Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Rockville, Maryland: Computer Science Press, 1978.
47. iPSC/2 User's Guide. Intel Corporation Publication. March 1988.
48. Ishida, Toru and Salvatore J. Stolfo. "Towards the Parallel Execution of Rules in Production System Programs." In *Proceedings of the International Conference on Parallel Processing*, pages 568-575, 1985.
49. Jackson, Peter. *Introduction to Expert Systems*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1986.
50. Labhart, Jay. Merit Technology, Leader, Merit Enhanced Traverse Engine for BBN Butterfly Development Team. Telephone interview. Merit Technology Inc., Plano TX.

51. Labhart, Jay, Steven Mattingly and Steven Carrow. "Forward Chaining Parallel Inference." In *Proceedings of the National Aeronautics and Electronics Conference*, Volume 1, pages 189-199, IEEE, 1990.
52. Laffey, Thomas J. and others. "Real-Time Knowledge Based Systems," *AI Magazine*, 9(1):27-45 (Spring 1988).
53. Lillevik, Sigurd. "Touchstone Program Overview." In *Proceedings of the Fifth Conference on Distributed Memory Computing*, pages 647-657, ACM, May 1990.
54. Luger, George F. and William A. Stubblefield. *Artificial Intelligence and the Design of Expert Systems*. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc., 1989.
55. Manna, Zohar. *Mathematical Theory of Computation*. New York, New York: McGraw-Hill Book Company, 1974.
56. McDowell, Charles E. and David P. Helmbold. "Debugging Concurrent Programs," *ACM Computing Reviews*, 21(4):593-622 (December 1989).
57. McNulty, Christa. "Knowledge Engineering for Piloting Expert System." In *Proceedings of the IEEE National Aerospace and Electronics Conference*, pages 1326-1330, IEEE, May 1987.
58. Milnes, Brian. The Production System Machine Project Member. Telephone interview. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 20 July 1989.
59. Ofizer, Kemal. "Partitioning in Parallel Processing of Production Systems." In *Proceedings of the International Conference on Parallel Computing*, pages 92-100, IEEE, 1984.
60. Ofizer, Kemal. *Partitioning in Parallel Processing of Production Systems*. MS thesis, Carnegie-Mellon University, Pittsburgh PA, March 1987.
61. O'Reilly, Cindy and Andrew S. Cromarty. "Fast is not Real-Time: Designing Effective Real-Time AI Systems." In *Proceedings of the International Conference for Optical Engineering*, pages 249-257, SPIE, April 1985.
62. Oshisanwo, A. O. and P. P. Dasiewicz. "A Parallel Model and Architecture for Production Systems." In *Proceedings of the International Conference on Parallel Computing*, pages 147-153, IEEE, 1987.
63. Pearl, Judea. *Heuristics, Intelligent Search Strategies for Computer Problem Solving*. Reading MA: Addison-Wesley Publishing Co., 1984.
64. Rich, Elaine. *Artificial Intelligence*. New York, NY: McGraw-Hill Co., 1983.
65. Riley, Gary. *Implementation of an Expert System Shell on a Parallel Computer*. Technical Report, NASA/Johnson Space Center, 1988. Houston, Texas.
66. Sabot, Gary W. *The Parallation Model; Architecture Independent Programming*. Cambridge MA: MIT Press, 1988.
67. Sawyer, George A. Static Methods for Optimum Rule Partitioning, Air Force Institute of Technology, March 1990.

68. Sawyer, George A. *Extraction and Measurement of Multi-Level Parallelism in Production Systems*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1990.
69. Sawyer, George A., R. Andrew Beard and Gary B. Lamont. A Compendium of Parallel Programs for Intel iPSC Hypercube Computers, Air Force Institute of Technology, March 1990.
70. Shakley, Capt Donald J. *Parallel Artificial Intelligence Search Techniques for Real-Time Applications*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1987.
71. Sommerville, Ian. *Software Engineering, Third Edition*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1988.
72. Stolfo, Salvatore J. and Daniel P. Miranker. "The DADO Production System Machine," *Journal of Parallel and Distributed Computing*, 3(2):269-296 (June 1986).
73. Stone, Harold S. *High-Performance Computer Architecture*. Reading, Massachusetts: Addison-Wesley, 1987.
74. Tambe, Milind. Production System Machine (PSM) Project Member. Telephone interview. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 14 Sep 1990.
75. Tambe, Milind, Anurag Acharya and Anoop Gupta. *Implementation of Production Systems on Message Passing Computers: Techniques, Simulation Results and Analysis*. Technical Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1989. Contract F33615-87-C-1499.
76. Tambe, Milind, Dirk Kalp, Anoop Gupta, Charles Forgy, Brian Milnes and Allen Newell. "Soar/PSM-E: Investigating Parallelism in a Learning Production System." In *Proceedings of the Conference on Parallel Programming Environments*, pages 146-161, IEEE, 1988.
77. Tenerio, M. F. M. and D. I. Moldovan. "Mapping Production Systems into Multi-processors." In *Proceedings of the International Conference on Parallel Computing*, pages 56-62, IEEE, 1985.
78. Whelan, Michael. Pilot's Associate: Approaching Maturity. Presentation to the 7th annual Workshop on Command and Control Decision Aiding, April 1990.
79. Wirth, Niklaus. *Algorithms + Data Structures = Programs*. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1976.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 1990		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE EXTRACTION AND EVALUATION OF MULTI-LEVEL PARALLELISM IN PRODUCTION SYSTEMS			5. FUNDING NUMBERS	
6. AUTHOR(S) George A. Sawyer, Capt, USAF				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCE/ENG/90D-04	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Attempts to apply "traditional" programming practices to "intelligent" decision and control problems (such as flying an aircraft) have typically failed due to the complexity of the associated computer code. Domain specific reasoning methods embodied by expert systems are successfully addressing many of the problems related to implementing "intelligent" control. The production system paradigm is one of the most common methods for implementing expert systems applications. However, one of the most significant problems related to the use of production systems is their relatively slow execution speeds. Past research shows that increases in execution speed through the use of parallel computer architectures are possible. However, these increases have been limited in scope (typically less than 10 fold) and difficult to achieve. This research examines the production systems to parallel computer architecture "transformation" from a theoretical basis. This approach provides significant insight into the salient factors that effect the efficient parallel implementation of production systems applications. The results indicate that the performance problem needs to be approached from both the application design and parallel algorithm design aspects. Additionally, this research shows that the level of parallelism that can be extracted from a production system application may be multiplicatively increased through the use of multi-level parallelism (combining parallel decomposition approaches). The use of multi-level parallelism offers the potential to increase the execution speed of production systems applications one or more orders of magnitude in comparison with existing production systems implementations on parallel computer architectures.				
14. SUBJECT TERMS Parallel Processing, Artificial Intelligence, Expert Systems, Production Systems			15. NUMBER OF PAGES 278	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	